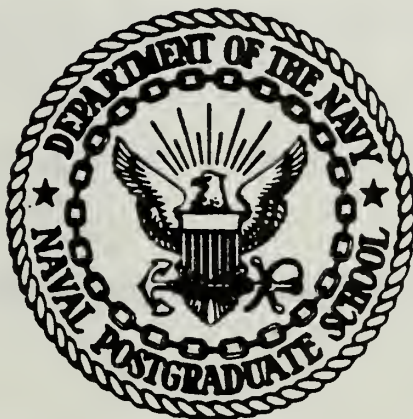


IMPLEMENTATION OF SEGEMENT MANAGEMENT
FOR A SECURE ARCHIVAL STORAGE SYSTEM

John Timothy Wells

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

IMPLEMENTATION OF SEGMENT MANAGEMENT
FOR A SECURE ARCHIVAL STORAGE SYSTEM

by

John Timothy Wells

September 1980

Thesis Advisor:

R. R. Schell

Approved for public release; distribution unlimited

T197055

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER		2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Implementation of Segment Management for a Secure Archival Storage System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; September 1980	
7. AUTHOR(s) John Timothy Wells		6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		8. CONTRACT OR GRANT NUMBER(s)	
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE September 1980	
		13. NUMBER OF PAGES 242	
		15. SECURITY CLASS. (of this report) UNCLASSIFIED	
		16. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)			
18. SUPPLEMENTARY NOTES			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) security kernel, operating systems, segmentation, information security, archival storage, security policy			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This thesis presents an implementation of segment management for the security kernel of a secure archival storage system. The basis for this implementation is a family of secure, distributed, multi-microprocessor operating systems designed to provide multi- level internal computer security and controlled sharing of data among authorized users. This implementation provides address space management for individual processes, based on segmentation as a			

memory management scheme. Non-discretionary information security is provided through enforcement of a security policy based on a lattice structure that allows flexibility in representing different security policies; the Department of Defense (DoD) security classification system is the security policy represented in this thesis. Implementation was completed on the ZILOG Z8000 micro-processor.

Approved for public release; distribution unlimited.

Implementation of Segment Management
for a
Secure Archival Storage System

by

John T. Wells
Lieutenant Commander, United States Navy
B.S., University of Mississippi, 1970

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1980

Therms
W 441
c. 1

ABSTRACT

This thesis presents an implementation of segment management for the security kernel of a secure archival storage system. The basis for this implementation is a family of secure, distributed, multi-microprocessor operating systems designed to provide multilevel internal computer security and controlled sharing of data among authorized users. This implementation provides address space management for individual processes, based on segmentation as a memory management scheme. Non-discretionary information security is provided through enforcement of a security policy based on a lattice structure that allows flexibility in representing different security policies; the Department of Defense (DoD) security classification system is the security policy represented in this thesis. Implementation was completed on the ZILOG Z8000 microprocessor.

TABLE OF CONTENTS

I.	INTRODUCTION.....	10
A.	BACKGROUND.....	11
B.	SECURE ARCHIVAL STORAGE SYSTEM OVERVIEW.....	14
1.	Levels of Abstraction.....	14
2.	Level Three - Host Computer(s).....	15
3.	Level Two - Supervisor.....	17
4.	Gate Keeper Module.....	19
5.	Level One - Kernel.....	21
a.	Segment Manager.....	21
b.	Non-Discretionary Security Module.....	22
c.	Event Manager.....	23
d.	Traffic Controller.....	24
e.	Inner Traffic Controller.....	25
f.	Memory Manager.....	28
6.	Level Zero - Hardware.....	31
C.	STRUCTURE OF THE THESIS.....	32
II.	SEGMENT MANAGEMENT FUNCTIONS.....	36
A.	BASIC CONCEPTS/DISCUSSION.....	36
1.	Segmentation.....	36
2.	Data Sharing.....	37
3.	Information Security.....	39
a.	Basic Security Principles.....	40
b.	Lattice Model Abstraction.....	43
c.	Examples.....	44

d.	Applications to the SASS.....	47
B.	SEGMENT MANAGER.....	48
1.	Function.....	48
2.	Database.....	50
C.	NON-DISCRETIONARY SECURITY MODULE.....	54
D.	MEMORY MANAGER.....	55
1.	Function.....	55
2.	Databases.....	56
E.	SUMMARY.....	58
III.	SEGMENT MANAGEMENT IMPLEMENTATION.....	60
A.	IMPLEMENTATION ISSUES.....	60
1.	Interprocess Messages.....	61
2.	Structures as Arguments.....	63
3.	Reentrant Code.....	63
4.	Process Structure of Memory Manager.....	64
5.	Per-Process Known Segment Table.....	64
6.	DBR Handle.....	65
B.	SEGMENT MANAGER MODULE.....	65
1.	Create a Segment.....	66
2.	Delete a Segment.....	69
3.	Make a Segment Known.....	70
4.	Make a Segment Unknown (Terminate).....	73
5.	Swap a Segment In.....	75
6.	Swap a Segment Out.....	75
C.	NON-DISCRETIONARY SECURITY MODULE.....	76
1.	Equal Classification Check.....	78

2. Greater or Equal Classification Check.....	78
D. DISTRIBUTED MEMORY MANAGER MODULE.....	80
1. Description of Procedures.....	80
2. Interprocess Communication.....	83
E. SUMMARY.....	85
IV. CONCLUSIONS AND FOLLOW ON WORK.....	98
APPENDIX A--SEGMENT MANAGER PLZ/SYS LISTINGS.....	100
APPENDIX B--SEGMENT MANAGER PLZ/ASM LISTINGS.....	110
APPENDIX C--DIST. MEMORY MANAGER PLZ/SYS LISTINGS.....	131
APPENDIX D--DIST. MEMORY MANAGER PLZ/ASM LISTINGS.....	141
APPENDIX E--NON-DISC. SECURITY PLZ/SYS LISTINGS.....	159
APPENDIX F--NON-DISC. SECURITY PLZ/ASM LISTINGS.....	161
APPENDIX G--SUMMARY OF REFINEMENTS.....	163
APPENDIX H--SEGMENT MANAGEMENT DEMONSTRATION.....	165
APPENDIX I--DEMONSTRATION LISTINGS.....	169
LIST OF REFERENCES.....	239
INITIAL DISTRIBUTION LIST.....	241

LIST OF FIGURES

1. SASS System Overview.....	16
2. Active Process Table.....	26
3. Virtual Processor Table.....	29
4. Summary of Extended Instruction Sets.....	34
5. Summary of Kernel Databases.....	35
6. Known Segment Table.....	53
7. Memory Management Unit Image.....	59
8. Memory Manager - CPU Table.....	59
9. Initialized Active Process Table.....	86
10. Initialized Virtual Processor Table.....	87
11. Initialized Known Segment Tables.....	88
12. Initialized Memory Management Unit Image.....	89
13. Initialized Process Stack Segments.....	90
14. Linker and Imager Command Lines.....	91
15. Load Command Lines and Register Initialization.....	92
16. Generated Output.....	93

ACKNOWLEDGEMENTS

This research is sponsored in part by the Office of Naval Research Project NR 337-005, monitored by Mr. Joel Trimble.

I wish to express my deepest appreciation to my advisor Lt. Col Roger Schell. His patience and assistance were invaluable. Thanks also to my reader Professor Lyle Cox and to Lcdr. Ed Moore, Lcdr. Steve Reitz, and Lt. Al Gary. Finally, special thanks to my closest friend (and wife) Susan, who always supports me, whatever the endeavor.

I. INTRODUCTION

This thesis addresses the implementation of the segment management functions of an operating system known as the Secure Archival Storage System or SASS. This system, with full implementation, will provide: (1) multilevel secure access to information (files) stored in a "data warehouse" for a network of multiple host computers, and (2) controlled data sharing among authorized users. The correct performance of both of these features is directly dependent upon the proper implementation of the segment management functions addressed in this thesis. The issue of access to sensitive information is addressed by the Non-Discretionary Security Module, which mediates all non-discretionary access to information. Sharing of information is accomplished chiefly through the properties of segmentation, the SASS memory management scheme that is supported by the Memory Manager Module and the Segment Manager Module. The implementation of segment management for SASS is thus integral to the attainment of the two key goals that SASS was designed to achieve. This implementation addresses the Non-Discretionary Security, Distributed Memory Manager (the interface to the Memory Manager Process), and Segment Manager modules.

A. BACKGROUND

O'Connell and Richardson provided the design for a family of secure, distributed, multi-microprocessor operating systems from which the subset, SASS, was later derived [6]. In their work, two of the primary motivations were to provide a system that (1) effectively coordinated the processing power of microprocessors and (2) provided information security.

The basis for emphasis on utilization of microprocessors is not purely that of replacing software with more powerful (and faster) hardware (microprocessors) but is also an economic issue. Software development and computing operations are becoming more and more expensive, putting further pressure on system designers to increasingly utilize people solely for system functions that computers cannot perform in a cost effective manner. Microcomputers, on the other hand, are becoming less and less expensive and are, therefore, increasingly being used for more functions.

The need for information security has been gradually recognized as the uses of computers have expanded. As security needs for specific computer systems have been recognized, attempts have been made to modify the existing systems to provide the desired security. The results have been systems that could not be certified as secure and/or which have failed to resist penetration efforts, i.e. systems which, in effect, did not provide adequate

information security. It has become clear that, in order to be certifiably secure, a computer system must have security designed in from first principles [10,11]. Such is the case with SASS. Information security was and continues to be a chief design feature. Integral to the design goal of information security were two related goals. One of these goals was to provide multilevel controlled access to a consolidated "warehouse" of data for a network of multiple host computers. The other key goal was to provide for controlled sharing among the computer hosts.

A brief background of prior work relative to SASS follows. O'Connell and Richardson originated the design of a secure family of operating systems. Their design provided two basic parts for their system -- the supervisor (to provide operating system services) and the kernel (to provide for physical resource management). The design of the SASS supervisor was completed by Parks [7]. No implementation or further design effort on the supervisor has followed, to date. The initial design of the kernel was completed by Coleman [1]. That design described the kernel in terms of seven modules:

1. Gate Keeper Module -- provided for ring-crossing mechanism and thus isolation of the kernel.
2. Segment Manager Module -- provided for management of segmented virtual memory.
3. Traffic Controller Module -- multiplexed processes onto virtual processors and supports the inter-

process communication primitives Block and Wakeup.
Block and Wakeup.

4. Non-Discretionary Security Module -- mediated non-discretionary security access attempts.
5. Inner Traffic Controller Module -- multiplexed virtual processors onto real processors and provided the Kernel synchronization primitives Signal and Wait.
6. Memory Manager Module -- managed main memory and secondary storage.
7. Input-Output Manager -- managed the moving of information to external devices outside the boundaries of the SASS.

Refinement of the kernel design and partial implementation was completed by Gary and Moore [4] in conjunction with Reitz [9]. The resultant description of the kernel as a result of their work was:

1. Gate Keeper Module
2. Segment Manager Module
3. Event Manager Module -- worked with the Traffic Controller to manage the "event data" associated with the IPC mechanism of eventcounts and sequencers.
4. Non-Discretionary Security Module
5. Traffic Controller Module -- replaced Block and Wakeup with Advance and Await (to implement Supervisor IPC mechanism of eventcounts and sequencers).
6. Memory Manager Module
7. Inner Traffic Controller Module

Reitz implemented the Traffic Controller Module and Inner Traffic Controller Module. Gary and Moore completed a

detailed design of the Memory Manager, originated the Memory Manager code (written predominantly in PLZ/SYS), selected a thread of the code, hand compiled it into PLZ/ASM and ran it on the Z8000 developmental module.

The design and implementation works mentioned above provided the design base for this implementation. Refinements were made as needed and are discussed in Appendix G of this thesis. A broader description of the current state of SASS will be provided in the next section.

B. SECURE ARCHIVAL STORAGE SYSTEM OVERVIEW

This section presents a brief summary of the current design state of the Secure Archival Storage System. The purpose of this summary is to provide continuity (interface) between this and previous work relative to SASS, and to enhance understanding of the evolution of the more detailed and system specific information provided later in this thesis.

1. Levels of Abstraction

The original design for a family of secure, distributed operating systems (which was the basis for the development of SASS) used effectively the concept of levels of abstraction as a design methodology tool. Just as this tool allows for clarity and simplification in conceptualizing and designing a system, it also enhances the ability to clearly and succinctly describe that system's

design. Thus, an abstract system overview (description) of SASS will be presented here. Figure 1 represents that overview (illustrated for a single host system for clarity). There are four levels of abstraction:

Level 3 -- the Host computer systems

Level 2 -- the Supervisor

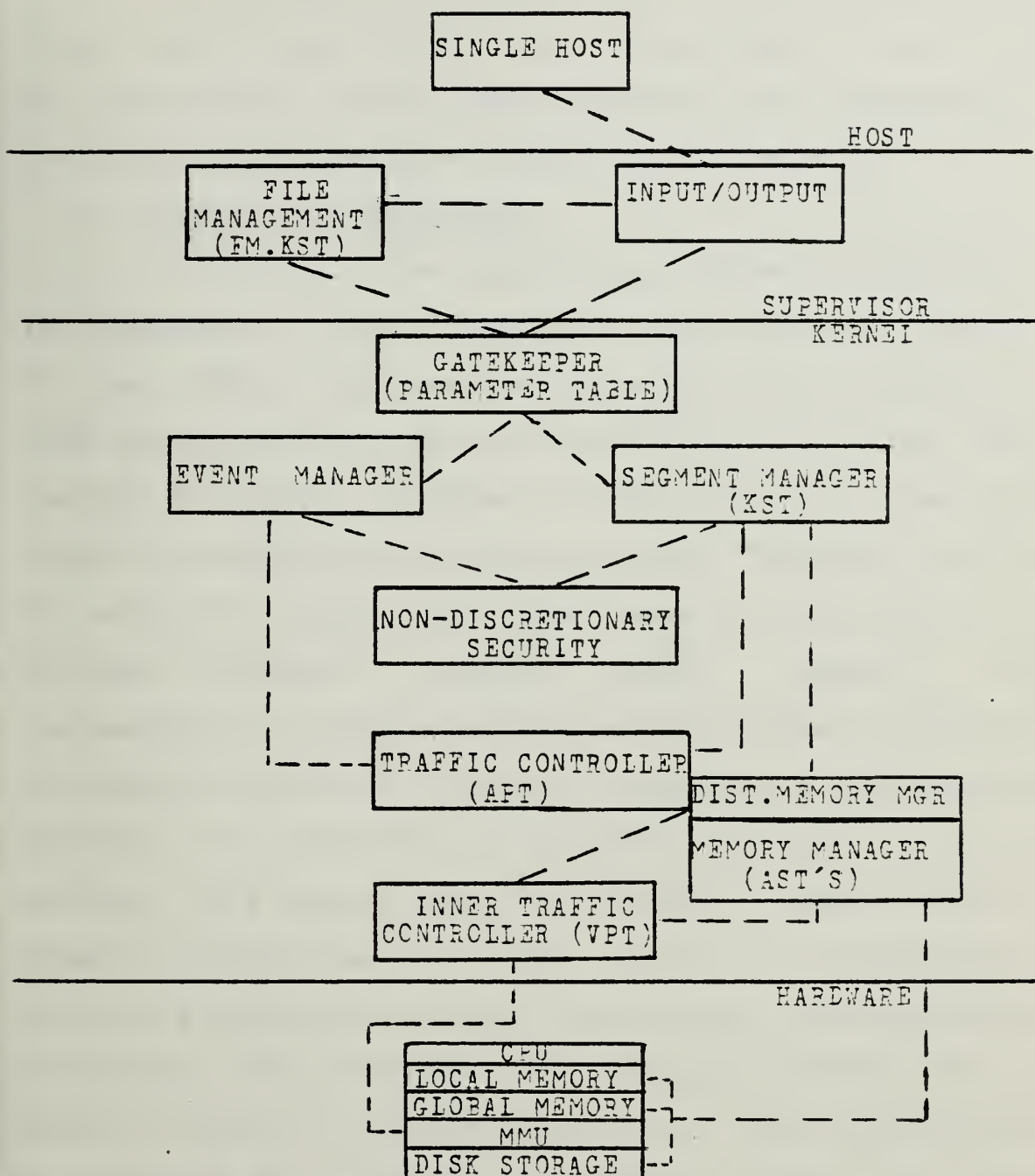
Level 1 -- the Kernel

Level 0 -- the Hardware

The Gate Keeper Module is logically the boundary between the Supervisor and the Kernel and thus will not be discussed within either of these levels, but rather separately.

2. Level Three - Host Computer(s)

Level three consists of the Host computer systems. There may be a variable number of host computers of any type (e.g., micro, mini, etc). Each host may be used to create and manipulate files of a fixed, predetermined degree of sensitivity (or security classification). Once a user of a host computer system completes work on a particular file, they can permanently store that file on the SASS (and, of course, may later again access the same file by requesting the SASS to provide it to them). Each host computer system is individually wired to an I/O port of the Z8001. Each of the ports has fixed access level.



Note: Databases are shown in parenthesis

Figure 1. SASS System Overview (Single Host)

If a multilevel secure Host desires to handle data at two levels (e.g., secret and unclassified), it will use two connections to the SASS. Physical and/or cryptographic protection of the hardwire connections is assumed.

3. Level Two - Supervisor

Level two is the Supervisor. A proper description of the Supervisor is "that component of the SASS that executes in the outer, less privileged domain (normal mode) of the Z8001 microprocessor, and is responsible for the SASS - Host computer interface". Integral to this description (and the Kernel's description) is the concept of a "domain", that can be described using four other terms that also need to be defined: "process", "address space", "segment", and "segmentation". Madnick and Donovan [5] define a process as the locus of points of a processor executing a collection of programs; the collection of programs and data that are accessed in a process forms that process's address space. A segment is defined as a logical grouping of information, such as a subroutine, array, or data area, and segmentation is defined as the technique for managing segments of an address space. It is convenient then, since the SASS uses segmentation as a memory management scheme, to more specifically define a SASS process address space as the collection of segments that are accessed (or are accessible) in that process. A domain is conceptualized in SASS due to the necessity to isolate the Kernel from all possible

outside influences. To achieve this, a process' address space is divided into a hierarchial arrangement of segment accessibility, viz., a set of hierarchial protection domains called protection rings. In SASS, there are two domains implemented (and necessary as a minimum): the Supervisor domain and the Kernel domain. The Z8001 microprocessor provides the SASS with two execution modes that, along with Kernel software, implement these domains: a system (Kernel) mode that provides access to all segments (and machine instructions), and a normal (Supervisor) mode that provides access to a subset of the segments (and machine instructions). Thus, the Supervisor operates in the outer or less privileged domain.

The Supervisor contains those segments of the system that are necessary to perform the SASS - Host computer system interface (construct and manage a file hierarchy and control I/O between the SASS - Host). It is built upon the Kernel and performs the Host's requests by calls to the Kernel (the calls are validated by the Gate Keeper prior to invocation of Kernel functions). Two surrogate processes, input/output (I/O) and file management (FM), are assigned to each Host computer system at system generation. The FM process directs all interaction between the SASS and a Host computer system. Specific functions include the management of the Host's file hierarchy (using the FM Known Segment Table (FM_KST) as a database) and discretionary security

access management (checking and maintaining an Access Control List (ACL) for each file within the file hierarchy). Controlling discretionary security with an ACL allows authorized users to specify who may use segments (files) within the confines of the non-discretionary security policy. Discretionary security will be defined and discussed in more detail in chapter II.

The I/O process acts in a slave mode to the FM process and is responsible for all input/output between the Supervisor and the host computer systems. Data is transferred between the Host and the SASS via fixed size "packets" (a grouping of data in a specified format). To transmit and receive packets between the Host and the SASS a "protocol" (or formal passing method) must exist between them. The I/O process is responsible for the SASS-Host protocol (Parks [7] designed a multi-packet protocol). Parks provides a detailed description of the Supervisor as it was originally designed.

4. Gate Keeper Module

The Gate Keeper is a software ring-crossing mechanism that provides for the isolation of the Kernel (viz., making the Kernel procedures tamperproof). The notion of a "ring-crossing" mechanism is an extension of the previous discussion of domains since "protection rings" is simply another term for hierarchial domains (such as the SASS arrangement of the Kernel and the Supervisor). All

calls to the distributed kernel and IPC with the Memory Manager must pass through the Gate Keeper (viz., it is the sole entry point into the Kernel from the Supervisor). The Gate Keeper is a trap handler; when invoked by the supervisor domain of a process, it must save the supervisor domain registers and stack pointer. The argument list provided by the supervisor domain's call (included in this list must be the identity of the kernel domain function (procedure) being called) is validated and, if correct, results in invocation of the appropriate procedure. Hardware preempt interrupts are masked upon entry into the Kernel. When returning (exiting the Kernel) the following actions occur: (1) software virtual preempt interrupts are unmasked (if a virtual preempt interrupt has occurred, the Traffic Controller's virtual interrupt handler is called vice the Kernel being exited), (2) hardware interrupts are unmasked, (3) the return arguments are passed to the Supervisor, and (4) the Supervisor domain stack pointer and registers are restored, returning the execution point to the Supervisor domain. An error code is returned and the Kernel is not invoked when an invalid call is encountered by the Gate Keeper. The database of the Gate Keeper is the Parameter Table. This table contains an entry for each permitted kernel function (e.g., Create_Segment, Delete_Segment, etc.) and is used to validate the correctness of the range (size) of the parameters passed.

5. Level One - Kernel

Level one is the Security Kernel (or Kernel). The Security Kernel in the inner or most privileged domain (system mode of the Z8001) and is responsible for managing the real resources of the hardware system (viz., memory, microprocessor, external devices, and input/output ports), and for enforcing the non-discretionary security policy for the SASS. The Kernel is divided into two major components. The first is the distributed kernel, i.e., the modules in the Kernel whose segments are placed in (or distributed in) the address spaces of each Supervisor process; the distributed kernel consists of the Gate Keeper (already discussed), the Segment Manager, the Event Manager, the Traffic Controller, and the Inner Traffic Controller. The second component is the non-distributed kernel and consists of the asynchronous memory manager process (which is contained entirely within the Kernel address space). There is a memory manager process for each hardware processor in the SASS. The following section will identify and briefly describe each of the Kernel's distributed and non-distributed system components.

a. Segment Manager

The Segment Manager (the focal point of this thesis) is a component of the distributed kernel; its function is the creation and management of a segmented virtual memory for the process. Actual memory management

functions are completed via calls for IPC to the Memory Manager process. Calls to (viz., entries into) the Segment Manager are received via the Gate Keeper from the Supervisor. These entries (viz., extended instructions) are:

1. Create_Segment -- add a new segment to the SASS.
2. Delete_Segment -- remove a segment from the SASS.
3. Make_Known -- add a segment to a process' address space.
4. Terminate -- remove a segment from a process' address space.
5. SM_Swap_In -- move a segment from secondary storage to main memory.
6. SM_Swap_Out -- move a segment from main memory to secondary storage.

The process local database used by the Segment Manager is the Known Segment Table (KST). The KST contains entries for all segments in the address space of that process. The Segment Manager will be described in more detail in chapters II and III.

b. Non-Discretionary Security Module

The Non-Discretionary Security (NDS) Module is a component of the distributed kernel; its function is the enforcement of the non-discretionary security policy in effect in the SASS. Although the implementation presented in this thesis reflects the DoD non-discretionary security policy, any security policy that can be represented by a

lattice structure may be similarly implemented. To implement a different policy (e.g., Privacy Act or a local policy) requires only replacement of the Non-Discretionary Security Module (viz., the modules calling it can be left intact with no changes required to them). The NDS Module creates the extended instruction set CLASS_EQ and CLASS_GE. The Non-Discretionary Security Module and the information security concepts which form its basis will be discussed in more detail in chapters II and III.

c. Event Manager

The Event Manager is a component of the distributed kernel; it is invoked by the Supervisor processes via the Gate Keeper. This module's function is to manage event data. Event data is associated with a global object (called an eventcount). An eventcount is a count of the number of events (e.g., the number of read or write accesses of a segment) that have occurred so far in the execution of a system. In SASS, as a naming convention, each Supervisor segment has two eventcounts associated with it. These eventcounts (Instance1 and Instance2) are stored in a Memory Manager database. The Event Manager creates the extended instruction set READ and TICKET; they are based on the mechanism of eventcounts and sequencers (used for the synchronization of concurrent processes). READ is a call that returns the current value of the eventcount. TICKET, using a nondecreasing integer called a sequencer (also

associated with each Supervisor segment), provides a complete time ordering of possibly concurrent events. Each invocation of the function TICKET increments the value of the sequencer and returns it to the caller. The eventcounts/sequencer synchronization mechanism is described in detail by Reed and Kanodia [8] while an excellent abridged discussion is presented by Gary and Moore [4].

d. Traffic Controller

The Traffic Controller (TC) is a component of the distributed kernel; it is responsible for multiplexing processes onto virtual processors. A virtual processor is a data structure that contains a complete description of a process in execution on a physical processor at a given instant. A "complete description" is defined to consist of the execution point or current CPU state and the address space (set of segments accessible by that process) of the process in execution. The Traffic Controller also creates the extended instructions ADVANCE and AWAIT which are used to implement eventcounts and sequencers, the inter-process communication (IPC) mechanism invoked by the Supervisor, and the extended instruction PROCESS_CLASS. PROCESS_CLASS is invoked by the Segment Manager and returns the label (classification) of the current process. The Traffic Controller is half of a two level traffic controller; the other half is the Inner Traffic Controller, which multiplexes the virtual processors onto physical processors.

The database for the Traffic Controller is the Active Process Table (APT), a fixed size, system wide database, that contains a permanent entry for each Supervisor process created at system generation (in the SASS the processes are then active for the life of the system). The APT structure is shown in figure 2. The scheduling algorithms and a detailed discussion of the current state of the Traffic Controller are provided by Reitz [9].

e. Inner Traffic Controller

The Inner Traffic Controller (ITC) is a component of the distributed kernel; it is the other half of the two level traffic controller and its function is to multiplex (temporarily bind) virtual processors (VP) to the real processors of the system; a design choice was made to provide each system CPU with a small fixed set of virtual processors. Two of the VP's are the Memory Manager VP and the Idle VP (the latter is permanently bound to the lowest priority virtual processor and is scheduled by the ITC only when there is no useful work for the CPU). The remaining VP's have Supervisor processes temporarily bound on them by the Traffic Controller. Another function of the ITC is to furnish inter-process services for VP's in the kernel ring. This is done by providing the primitives SIGNAL and WAIT, that are used by processes in the Kernel ring to communicate with other Kernel ring processes.

LOCK	
RUNNING LIST	PROCESS ID
VP_ID →	
	:
READY LIST	
BLOCKED LIST	

AP_INDEX

DBR	ACCESS_CLASS	STATE	NEXT_AP	EVENTCOUNT HANDLE INSTANCE COUNT

Figure 2. Active Process Table

APT	RECORD	[LOCK	WORD
		RUNNING_LIST	RUNNING_ARRAY
		READY_LIST	WORD
		BLOCKED_LIST	WORD
		AP	ARRAY
		[NR_PROCESSES	AP_TABLE]
]	
AP_TABLE	RECORD	[NEXT_AP	AP_POINTER
		DER	ADDRESS
		ACCESS_CLASS	LONG
		STATE	INTEGER
		NEXT_AP	WORD
		EVENT_COUNT	EVENT_TABLE
]	
EVENT_TABLE	RECORD	[HANDLE	WORD
		INSTANCE	WORD
		COUNT	WORD
]	
AP_POINTER	WORD		
ADDRESS	WORD		

Figure 2. Active Process Table (continued)

This is the mechanism used within the Kernel to provide multiprogramming (process switching). The ITC Module creates the extended instruction set: SIGNAL, WAIT, SWAP_VDBR, IDLE, SET_PREEMPT, TEST_PREEMPT, and RUNNING_VP. The functions of SIGNAL and WAIT have been discussed already. SWAP_VDBR provides the TC with a means to schedule processes on the currently running VP. IDLE loads an "idle" process on the currently running VP. SET_PREEMPT sets the virtual preempt interrupt flag on a specified VP (specified by the TC). TEST_PREEMPT provides the virtual preempt unmasking mechanism that is executed each time a process tries to move from the Kernel to the Supervisor domain. The database used by the Inner Traffic Controller is the Virtual Processor Table (VPT). There is one system wide table with entries for each physical processor in the system. The VPT for a single processor system (such as SASS) is shown in figure 4. The scheduling algorithms and a detailed discussion of the current state of the Inner Traffic Controller are provided by Reitz[9].

f. Memory Manager

The Memory Manager Module is the only component in the non-distributed kernel. There is a Memory Manager process dedicated to each physical processor (CPU) in the system.

LOCK	
RUNNING_LIST	
READY_LIST	
FREE_LIST	

VP_INDEX

DBR_NO	PRI	STATE	IDLE_FLAG	PREEMPT	PHYS_PROCESSOR	NEXT_VP	MSG_LIST

MSG_INDEX

MESSAGE	SENDER	NEXT_MSG

Figure 3. Virtual Processor Table

VPT	RECORD	[LOCK RUNNING_LIST READY_LIST FREE_LIST VP MSG_Q	WORD VP_INDEX VP_INDEX MSG_INDEX ARRAY [NR_VP VP_TABLE] APRAY [NR_VP MSG_TABLE]
VP_TABLE	RECORD	[DBR PRI STATE IDLE_FLAG PREEMPT PHYS_PROCESSOR NEXT_READY MSG_LIST]	ADDRESS WORD WORD WORD WORD WORD VP_INDEX MSG_INDEX
MESSAGE	ARRAY	[16	BYTE]
ADDRESS	WORD		
VP_INDEX	INTEGER		
MSG_INDEX	INTEGER		

Figure 3. Virtual Processor Table (continued)

The Memory Manager is responsible for managing the real memory resources of the system, viz., local and global main memory and secondary storage. The memory manager manages the local and global memory in such a way as to control bus contention in the multi-microprocessor environment. Thus, each CPU has its own local memory to store process local segments and there is a global memory to which every CPU has access and in which shared, writeable segments must be stored. This requirement is to ensure that a current copy is always accessed for a shared, writeable segment. To keep bus contention between processors that access global memory to a minimum, whenever possible (viz., in all cases but shared, writeable segments) segments are to be stored in local memory. The Memory Manager has several databases, primary of which are the system wide Global Active Segment Table (G_AST) and the per processor Local Active Segment Table (L_AST). A more detailed description of the Memory Manager Module is presented in chapters II and III.

6. Level Zero - Hardware

The Z8001 microprocessor, Z8010 Memory Management Unit (MMU), local and global memories, and secondary storage form the SASS' basic hardware group. Since the design calls for SASS to exist in a multi-microprocessor environment, there will be multiple copies of some elements of the group, e.g., CPU, local memory. The Z8001 microprocessor is a

register oriented machine that has sixteen 16-bit general purpose registers. When operated with the MMU, the desired capabilities of memory segmentation, multiple domains, and process switching are realized. The MMU consists of a set of registers (64) to implement the descriptor list (or descriptor segment); viz., each register contains the descriptor (containing the attributes) of a particular segment. Zilog [14] provides a detailed description of the Z8001 microprocessor and Zilog [15] describes the Z8010 MMU.

C. STRUCTURE OF THE THESIS

This thesis describes the implementation of the segment management functions for the SASS. The design "base" evolved from the original secure family of operating systems identified and designed by O'Connell and Richardson. A block structured language, PLZ/SYS, was used in this and previous design efforts, while implementation was completed using PLZ/ASM assembly code. PLZ/SYS is described by Snook [12] and Conway [2] while PLZ/ASM is described by Zilog [13]. A compiler for PLZ/SYS to PLZ/ASM code translation was not available. As a result, implementation included the added step of manual translation of PLZ/SYS code to PLZ/ASM code to facilitate testing and debugging.

In this chapter an introduction to SASS was provided through discussion of its background and an overview of the entire system. A summary of the extended instruction sets

created by the Kernel components and a summary of the Kernel databases is presented in figures 4 and 5.

Chapter II of this thesis will present a description of the segment management functions in SASS. Discussion of the theory behind information security and its implications to SASS is also provided. The modules encompassed by segment management will be discussed in terms of their design, functional purpose, and database descriptions.

Chapter III presents the implementation of segment management (viz., the segment manager, non-discretionary security, and distributed memory manager modules). Description of design and implementation criteria, and choices made during implementation are discussed in this chapter.

Chapter IV provides the conclusions reached, status of research, and recommendations relative to continuation and extension of the work.

Appendices include PLZ/ASM code for the modules, the program listings for the Segment Manager demonstration and a summary of the refinements made to previous design/code relative to SASS.

MODULE	INSTRUCTION SET	
Segment Manager	Create_Segment	Delete_Segment
	Make_Known	Terminate
	SM_Swap_In	SM_Swap_Out
Event Manager	Read	Ticket
Non-Discretionary Security Traffic Controller	Class_EQ	Class_GE
	Advance	Await
	Process_Class	
Inner Traffic Controller	Signal	Wait
	Swap_VDBR	Idle
	Set_Preempt	Test_Preempt
	Running_VP	
Memory Manager	MM_Create_Entry	MM_Delete_Entry
	MM_Activate	MM_Deactivate
	MM_Swap_In	MM_Swap_Out

Figure 4. Extended Instruction Sets

MODULE	DATABASE
Gate Keeper	Parameter Table
Segment Manager	Known_Segment_Table (KST)
Traffic Controller	Active_Process_Table (APT)
Inner Traffic Controller	Virtual_Processor_Table (VPT)
Memory Manager	Global_Active_Segment_Table (G_AST)
	Local_Active_Segment_Table (L_AST)
	Memory_Management_Unit_Image (MMU_Image)
	Alias_Table
	Disk_Bit_Map
	Global_Memory_Bit_Map
	Local_Memory_Bit_Map

Figure 5. Kernel Databases

II. SEGMENT MANAGEMENT FUNCTIONS

A conceptual discussion of the functions associated with segment management is presented in this chapter. As previously mentioned, two dominating goals of SASS were to provide multilevel controlled access to information and to provide for controlled sharing of information. The major factor in controlled access (which, in effect, refers to information security) and in information sharing is the concept of segmentation. Segmentation, data sharing, and information security will be discussed relative to their value to SASS.

A. BASIC CONCEPTS/DISCUSSION

1. Segmentation

Segmentation has previously been defined as the technique for managing segments of an address space where a segment is defined to be a logical grouping of information which possesses the qualities of having uniform attributes, being logical (vice physical), being visible to the user and being arbitrary in size. Based upon this notion, a process' address space is then viewed as consisting of the collection of segments that the process may access. In a segmented environment all addresses require two components: (1) a segment specifier (number) and (2) the location (offset) within the segment. Each segment may have attached to it

logical attributes that enable certain important control features to be implemented. Controlled access and information sharing implementation is specifically facilitated. By including classification and access information in a segment's logical attributes, a method to enforce information security is provided. Segmentation supports information sharing since it allows a segment to belong to more than one address space, that is, a single physical copy may be accessed by more than one process. Controlled physical sharing of information (within access constraints) is achieved, in the case of SASS, by putting segments which are shared and writeable into the system's global memory (vice a copy in each local memory).

Segmentation also facilitates the implementation of multiple protection domains in SASS. A process' address space is divided into domains or arrangements of segment accessibility. The Kernel domain is the most privileged and includes all segments of the address space, while the supervisor domain is less privileged and excludes segments representing the management of the shared resources by more than one process.

2. Data Sharing

The facility to share a single segment (and thus a single copy of the information to be accessed) by many processes is a significant feature that is facilitated via segmentation. In short, by processes sharing a common

physical copy of a segment, there is no requirement for duplicate copies and thus no possibility exists of having copies that are not up to date. In SASS, given the global memory/local memories environment, the policy is to put copies of segments in local memory except in the case of shared and writeable segments, which are placed in global memory for sharing purposes among processes with the appropriate access.

Segmentation is vital to this policy since only through explicit segmentation can SASS know the read/write properties of the information. Thus, segments which are shared but have read only access (by all processes that may access it) are not put into global memory but rather into the local memory of each of the processes that may access it (viz., multiple copies exist). There is no possibility of the multiple copy/ out of date copy problem since only read access is allowed. However, this is a seeming waste of memory and nonuse of the sharing facility provided by segmentation. The justification is based on a design decision motivated by another goal of SASS -- reduction of bus contention among processors accessing global memory. This is considered to be of more importance than the saving of memory space offered by single copy sharing of information; as stated before, the cost of memory has gone down significantly in the last few years thus reducing its influence on decisions such as this.

3. Information Security

Information security in a computer environment only recently began to receive the attention that it deserves. Few people have been far sighted enough to view computer security in an analogous manner to communications security (an area which has received considerable military and commercial attention throughout history, especially since the advent of electronic communications). Only through harsh and embarrassing lessons has the importance of computer security being recognized. The range of problems encountered covers virtual every level of computer usage: banks and commercial enterprises are victims of theft through the felonious use of computers; universities are the victims of undesired users entering their systems and either maliciously or accidentally destroying valuable programs; and the military faces the real possibility that classified material is being accessed by foreign agents without our knowledge and/or crucial systems are being tampered with without our knowledge. The effects of these type actions may be as small as simple embarrassment or as serious as undermined military preparedness. It should be clear that information security is a serious issue. Definitively, this thesis will consider information security as the process of providing controlled access to information based on proper authorization. A pertinent information security goal is to provide a "multilevel" information security environment

(that is, an environment where multiple levels of sensitive information and user accessibility to that information exist together in a manner such that security is not compromised). The key to achieving computer security lies with the concept of the "security kernel". A discussion of this concept and some supporting definitions is provided in the next section.

a. Basic Security Principles

The protection of secure information in computer systems is affected through two types of control: (1) external controls -- where physical means are used to securely isolate the computer system (e.g., an armed guard) and (2) internal controls -- where the computer itself provides protection by distinguishing information security levels and user accessibilities. Although the discussion in this thesis centers around internal controls, external controls are also a viable and important aspect of the SASS (and other computer system's) information security. As previously stated, the key (or answer) to computer security lies in the security kernel concept. Schell [11] provides a detailed development of the theory behind this concept. The security kernel is defined as that part of the computer system's hardware and software which enforces the authorized access relationships between the user/process (subject) and the accessed information (segment or object).

An important aspect to the development of a secure kernel based system is the security policy to be

enforced. There are two distinct aspects of security policy. The first is the non-discretionary (mandatory) policy that externally constrains what access is permissible; this policy is manifested and implemented in an arrangement where information in the form of a segment (called an "object") is labelled as to its sensitivity; the same is done with the party requesting access (the user/process, called a "subject"). The relationship between the subject and object "labels" that leads to an access permission or denial is defined by a lattice structure [3]. This lattice structure concept will be discussed in the next section. The second aspect of security policy is the discretionary policy, which is a refinement within the non-discretionary constraints. It is emphasized that discretionary security is contained within (and in no way substitutes for) non-discretionary security. An example is the "need to know" policy of the DoD. Implementation of the discretionary security policy for SASS is accomplished in the Supervisor through the maintenance of an Access Control List (ACL) for each file in the file hierarchy. Each access attempt to a file is checked against the ACL and access is granted in accordance with that check and the non-discretionary security check (whichever granted the least access). This allows the users to specify (subject to non-discretionary security constraints) who may access their files. Since the implementation of the discretionary security policy is not a

part of this thesis, a detailed discussion is not provided. Parks [7] provides a discretionary security policy design for SASS.

Implementation of a security policy requires an awareness of and consideration for several basic security properties which are briefly defined below.

The Simple Security Condition restricts a subject's read access to objects whose classification is equal to or less than than the subject's classification (the term classification will hereafter be used to indicate a degree of sensitivity or security importance).

The Confinement Property (or "*-property") restricts a subject's write access to objects whose classification is equal to or greater than the subject's classification. This property prevents a subject from writing to an object of lower classification where another subject (of less than the original subject's classification) would have potential access thus violating security.

The Compatibility property has as a basis the hierarchial structure of the objects (segments) of SASS. The objects of SASS are hierarchially organized in a tree structure. The structure consists of nodes, leaves, and a root from which the tree emanates. A node (an alias table that contains a list of attributes for segments) is directly associated with a segment that is the "mentor" for one or more segments. A leaf, viz., a segment, is not an alias

table but may be a mentor segment (with the same access class as the alias table). The Compatibility property basically states that the object access classification must be non-decreasing in moving from the root down the hierarchy (viz., the access classification of a child must be greater than or equal to its parent).

b. Lattice Model Abstraction

A lattice model of secure information flow concept (discussed by Denning [3]) permits concise formulations of the security requirements of different systems and facilitates the construction of mechanisms that enforce security. Specifically, the relationship between classifications can be represented by a partially ordered lattice structure (examples illustrating this concept are presented in the next section). Authorizations for access (or decisions on compatibility) then are based on this lattice. With the properties previously discussed as a basis, the accesses permitted are defined below ("sac" is the abbreviation for "subject access classification" and "oac" for "object access classification" and "|" denotes "not related"):

1. $sac = oac$, read/write permitted
2. $sac > oac$, read permitted
3. $sac < oac$, write permitted
4. $sac \mid oac$, no access permitted

Case 3 represents a subject's ability to "write up", which is a capability not supported in SASS; thus for SASS, case 3 is more accurately represented as:

3. sac < oac, no access permitted

At this point, no design detail has been provided for the representation of a classification or for defining how two classifications are compared and determined to be "equal", "greater than", "less than", or "unrelated" (viz., what does sac>oac mean?). The next section will illustrate these ideas both generally and by examples.

c. Examples

Based on military influence, the examples provided are reflective of a subset of the DoD non-discretionary security policy. A classification (or label) is defined to have two parts: (1) a level (e.g., top secret, secret, confidential, or unclassified in the example) and a category (e.g. Crypto (Cy), Nato (N), Nuclear (Nu), or empty (%) in the example). In the actual implementation (chapter III), provisions will be made for eight levels and sixteen categories. (In some reference texts, levels are called categories and categories are called compartments). The levels are defined by a "totally ordered" relationship where all levels are related:

Top Secret > Secret > Confidential > Unclassified

or

TS > S > C > U

The categories are defined as "disjoint" (no relationship exists when comparing individual categories with other individual categories). The classifications (labels) (the concatenation of level with category) then are defined to have a "partially ordered" relationship since some but not all classifications are related. The cases to be illustrated will be illustrated through a general case and then by specific examples. The general structure is defined by:

Subject's classification = (LS, {CS})

Object's classification = (LO, {CO})

where:

LS = Subject's level

{CS} = Subject's set of categories

LO = Object's level

{CO} = Object's set of categories

The non-inclusive set of partially ordered examples will be chosen from a subset of the classifications derivable from the set of totally ordered levels and the set of disjoint categories:

{TS,S,C,U} and {Cy,N,Nu,%}

Case I : Equal (sac =oac)

General - LS = LO and {CS} = {CO}

Examples - (TS,{Cy,N}) = (TS,{Cy,N})

- (U, {Cy}) = (U ,{Cy})

Access - Read/Write

Case II: Greater than ($sac > oac$)

(1) General - $LS > LO$ and $\{CO\}$ subset to $\{CS\}$

Examples - $(S, \{N, Nu\}) > (C, \{N\})$

- $(S, \{N, Nu\}) > (U, \{\%\})$

(2) General - $LS > LO$ and $\{CS\} = \{CO\}$

Examples - $(TS, \{\%\}) > (S, \{\%\})$

- $(S, \{Nu, N\}) > (U, \{Nu, N\})$

(3) General - $LS = LO$ and $\{CO\}$ proper subset to $\{CS\}$

Examples - $(U, \{Nu\}) > (U, \{\%\})$

$(TS, \{Cy, N, Nu\}) > (TS, \{Cy\})$

Access - Read

Case III: Less than ($sac < oac$)

(1) General - $LS < LO$ and $\{CS\}$ subset to $\{CO\}$

Examples - $(S, \{\%\}) < (TS, \{N\})$

- $(U, \{N, Nu\}) < (C, \{Cy, N, Nu\})$

(2) General - $LS < LO$ and $\{CS\} = \{CO\}$

Examples - $(S, \{\%\}) < (TS, \{\%\})$

- $(U, \{N, Nu\}) < (C, \{N, Nu\})$

(3) General - $LS = LO$ and $\{CS\}$ proper subset to $\{CO\}$

Examples - $(U, \{\%\}) < (U, \{N\})$

- $(C, \{N, Cy\}) < (C, \{N, Nu, Cy\})$

Access = no access (in SASS)

= Write (in principle)

Case IV : Unrelated ($sac \mid oac$)

(1) General - $LS <, >, \text{or} = LO$ and $\{CS\} \mid \{CO\}$

Examples - (S, {N}) | (C, {Nu})
 - (C, {Nu}) | (S, {N})
 - (TS, {N}) | (TS, {Cy})

(2) General - LS > LO and {CS} proper subset to {CO}

Examples - (TS, {X}) | (S, {N})
 - (C, {N, Nu}) | (U, {N, Nu, Cy})

Explanation - there is a contradiction between
 the relationship of the levels and
 the relationship of the categories.
 Since this contradiction is
 unresolvable, the classification
 relationship must be "unrelated".

(3) General - LS < LO and {CO} proper subset to {CS}

Examples - (S, {N, Nu}) | (TS, {N})
 - (U, {Cy}) | (C, {X})

Explanation - same as above

Access = No access

d. Applications to the SASS

The cases above are designed to identify each possible relationship that exists between two labels. In the SASS, it is necessary only to identify cases I and II (label 1 >= label 2), while lumping the other cases into a single case which represents "no access". This arrangement encompasses enforcement of the Confinement Property, Simple Security Condition, and the Compatibility property. Enforcement must occur on every access attempt of an object.

A discussion of the implementation of non-discretionary security policy is provided in the next chapter.

B. SEGMENT MANAGER

1. Function

The Segment Manager is the focal point of the segment management function. Using the per-process Known Segment Table as its database and the Memory Manager and Non-Discretionary Security Module in strongly supportive roles, it is responsible for managing the segmented virtual memory for a process. Its role can be viewed as somewhat intermediary in nature (viz., between the Supervisor modules and the Memory Manager modules). The extended instruction set created in the Segment Manager includes the following instructions: CREATE_SEGMENT, DELETE_SEGMENT, MAKE_KNOWN, TERMINATE, SM_SWAP_IN, and SM_SWAP_OUT (note that the names for SWAP_IN and SWAP_OUT have been modified by preceding each with SM_; this is strictly for clarity because the Memory Manager also creates two instructions called SWAP_IN and SWAP_OUT). These instructions are invoked by the Supervisor domain of the process (viz., calls are made from the Supervisor domain via the Gatekeeper to the Segment Manager in the Kernel domain) to provide SASS support to the Host.

In general, when the Segment Manager receives these calls, it performs certain checks to ensure the validity and

security compliance (when required) of the request (call). These checks are performed using its own database (the KST) and by calls to the Non-Discretionary Security Module (when required). The Segment Manager invokes one of six Memory Manager (more specifically, the Distributed Memory Manager Module) created instructions. These instructions include: MM_CREATE_ENTRY, MM_DELETE_ENTRY, MM_ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. These invoked instructions (procedures) in turn perform interprocess communications with the non-distributed memory manager process (where actual memory management functions are accomplished). These interprocess invocations and returns are accomplished through the use of the IPC primitives Signal and Wait. The Segment Manager returns the required arguments to the Supervisor by value (as passed back to it by the Memory Manager and/or determined within itself). The Segment Manager performs actual segment number assignment when a segment is made known to a process' address space. It also performs any further database (KST) updating as may be required. A more detailed description of the specifics of the actions of the Segment Manager will be provided in the implementation described in Chapter III.

2. Database

The Known Segment Table (KST) is the database used to manage segments. The KST is described in its tabular form and PLZ/SYS structured representation in figure 6. There are several basic and pertinent facts to be noted of the KST:

1. It is a process local database; that is, each process has its own KST.
2. The KST is indexed by segment number; each record of the KST consists of a set of fields (description information) regarding a particular segment.
3. Entering information into the fields of a segment is called "making a segment known". This simply refers to adding a segment to a process' address space (viz., making a segment accessible to a process).
4. In SASS, a correspondence exists between making a segment "known" and making a segment "active"; i.e., when a segment is added to the address space of a process, this action results in an entry in the KST (making "known") by the Segment Manager and an entry in the Global Active Segment Table (G_AST) by the Memory Manager process (making it "active").

The G_AST will be described later in this chapter.

A proper description of the structure and fields of the KST is necessary at this point. Using the representation of the PLZ/SYS language structure, the KST is described as an array

of records of fields of varying types. The fields are described separately below. Although the KST index is not in itself a field in the record, it does perform a rather significant role. The KST index is an integer closely related to the segment number of the segment described in that KST entry (viz., it is the subscript into the array of records). This segment number also corresponds to the MMU descriptor register (number) for that segment.

The MM_Handle is the first field in a KST record. The MM_Handle is a system wide unique number that is assigned to each segment with an entry in the G_AST (viz., every active segment). This "handle" is the instrument of controlled single copy sharing of information (segments). It allows a segment to exist under one unique handle but be accessible in the address space of more than one process (with different segment numbers in each address space). The MM_Handle is returned to the Segment Manager by the Memory Manager during the execution of the Make_Known instruction.

The Size field is an integer value (of language structure type "word") which represents the number of 256 byte blocks composing a segment.

The Access_Mode field is used to describe the process' access to the segment (i.e., null or read and/or write).

The In_Core field is used to indicate if the segment is or is not in main memory (i.e., this field is a flag or true/false boolean switch).

The Class field is a long word field used to represent the degree of information sensitivity (viz., access class) assigned to the segment. This field (for example) would be used to numerically describe a classification label (as described above).

The Mentor_Seg_Nr field is a number representing the segment number of a segment's parent or "mentor" segment. Its importance will be discussed shortly.

The Entry_Nr field is a number representing a segment's index number into its parent or mentor segment's Alias Table (not yet discussed).

The Alias Table is a Memory Manager database and will be described later. The aliasing scheme provided via the alias tables is used to prevent passing system wide information out of the Kernel (i.e., the Unique_ID of a segment). The "alias" of a segment is the concatenation of the Mentor_Seg_Nr with the segment's Entry_Nr (index) into the mentor segment's Alias Table. It is clear that the last two fields of a KST record are the "alias" of that segment.

Segment_#

MM_Handle	Size	Access_Mode	In_Core	Class	M_Seg_No	Entry_Number

KST Array [64 KST_REC]

KST_REC Record [MM_Handle Array [3 Word]
 Size Word
 Access_Mode Byte
 In_Core Byte
 Class Long
 M_Seg_No Short_Integer
 Entry_Number Short_Integer]

Figure 6. Known Segment Table.

C. NON-DISCRETIONARY SECURITY MODULE

The key in protection of secure information using internal controls was identified as the security kernel concept. The basic idea within this concept is to prove the hardware part of the Kernel correct and, similarly, to keep the software part small enough so that proving it correct is feasible. A central component of the kernel software is the Non-Discretionary Security Module (hereafter referred to as the NDS Module). The NDS Module is concerned only with the non-discretionary aspect of the security policy in effect; since the discretionary aspect is subservient in nature to the non-discretionary aspect, it is then sufficient that the Kernel contain only the software representing the non-discretionary aspect of the security policy. The discretionary security is provided outside the kernel in the SASS supervisor. Every attempt to access information must result in an invocation of the NDS Module.

The function of the NDS Module is to compare two classifications (viz., compare two labels), make a decision as to their relationship (i.e., =, >, <, !), and return a true/false interpretive answer relative to the query of the calling procedure. The mechanism used as a basis is the lattice model abstraction previously discussed. The NDS Module does not require a database since the labels it compares are stored in (passed from) other Kernel databases.

D. MEMORY MANAGER

1. Function

The Memory Manager process is the only component of the non-distributed kernel. It is responsible for managing the real memory resources of the system -- main (local and global) memory and secondary storage. It is tasked by other processes within the Kernel domain (via Signal and Wait) to perform memory management functions. This thesis will address the Memory Manager in terms of two components: (1) the Memory Manager Process (also called the nondistributed kernel and the Memory Manager Module), and (2) the distributed Memory Manager (also called the Distributed Memory Manager Module). The former is the "true" memory manager while the latter is the interface with other processes, that is, it resolves the issue of interprocess communication with the "true" memory manager.

The Distributed Memory Manager Module creates the following extended instruction set: MM_CREATE_ENTRY, MM_DELETE_ENTRY, MM_ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. The instructions form the mechanism of communication between the Segment Manager of a process and a memory manager process (where the actual memory management functions are performed). The Memory Manager Process instruction set corresponds one to one with that of the Distributed Memory Manager; the set consists of: CREATE_ENTRY, DELETE_ENTRY, ACTIVATE, DEACTIVATE, SWAP_IN,

and SWAP_OUT. The basic functions performed by the Memory manager are allocation/deallocation of global and local memory and of secondary storage, and segment transfers from local to global memory (and vice-versa) and from secondary storage to main memory (and vice-versa).

2. Databases

A detailed and descriptive discussion of the Memory Manager databases is presented in the work of Gary and Moore [4] and the reader may refer to it for memory manager database details. This thesis addresses the implementation of the distributed Memory Manager but not the Memory Manager Process, thus brief descriptions are provided of the latter's databases.

The Global Active Segment Table (G_AST) is a system wide (i.e., shared by all memory manager processes) database used to manage all active segments. A lock/unlock mechanism is used to prevent race conditions from occurring. The distributed memory manager of the signalling process locks the G_AST before it signals the memory manager process.

The Local Active Segment Table (L_AST) is a processor local database which contains an entry for each segment active in a process currently loaded in local memory.

The Alias Table is a system wide database associated with each nonleaf segment in the Kernel. It is a product of the aliasing scheme used to prevent passing system wide

information out of the Kernel. The alias table header (provided for file system reconstruction after system crashes) has two pointers, one linking the alias table to its associated segment, the other linking the alias table to the mentor segment's alias table. The fields in the alias table are Unique_ID, Size, Class, Page_Table_Loc, and Alias_Table_Loc. The index into the alias table is Entry_No.

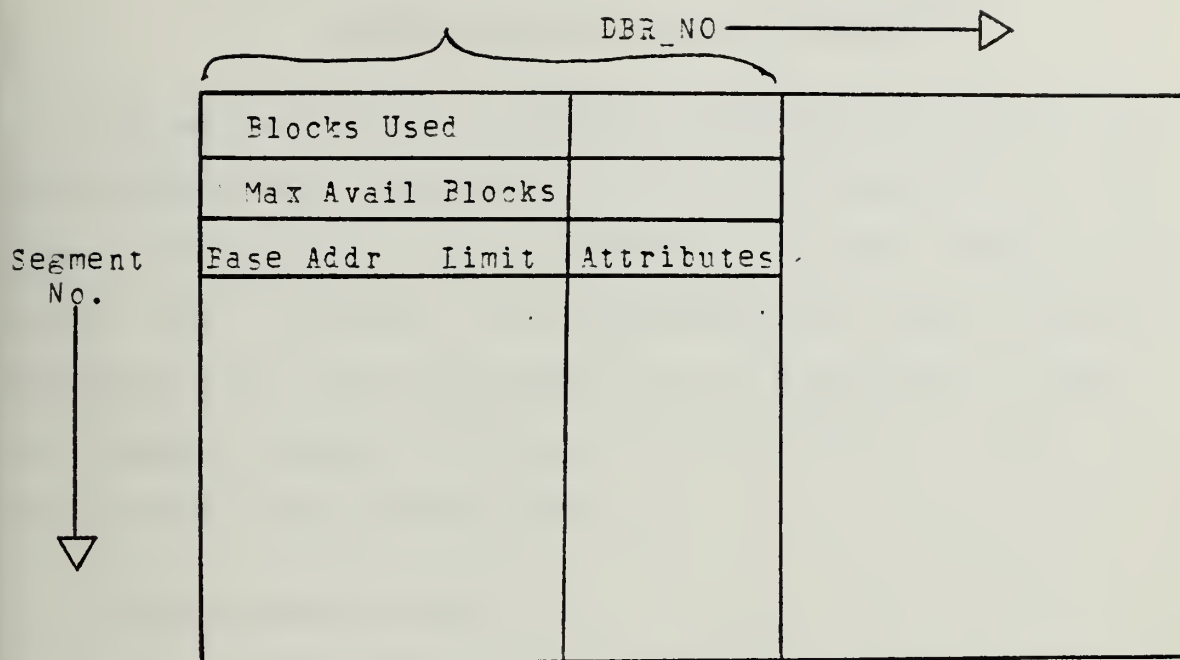
The Memory Management Unit Image (MMU_Image, figure 7) is a processor local database indexed by DBR_No (viz., for each DBR_No there is a MMU_Image record, with each record containing a software image of the segment descriptor registers of the hardware MMU). The MMU_Image is an exact image of the MMU. Each record is indexed by Segment_No (segment number) and each Segment_No entry contains three fields. The Base_Addr field contains the segment's base address in memory. The Limit field contains the number of blocks of contiguous storage for the segment (zero indicates one block). The Attributes field contains 8 flags including 5 which relate to the memory manager. The Elks_Used field and the Max_Blks (available) fields are per record (not per segment entry) and are used in the management of each process' virtual core.

The Memory Bit Maps (Disk_Bit_Map, Global_Memory_Bit_Map, and Local_Memory_Bit_Map) are memory block usage maps that use true/false flags (bits) to indicate the use or availability of storage blocks.

The only database in the Distributed Memory Manager is the Memory Manager CPU Table. It is an array of memory manager VP_ID's (MM_VP_ID) indexed by CPU number. This table enables a signalling process to identify the appropriate memory manager process (virtual processor) to signal.

E. SUMMARY

The segment management functions and key related concepts (such as segmentation) were discussed in this chapter. The importance of segmentation to data sharing and information security was emphasized as were key information security concepts. With this background, the implementation of segment management and a non-discretionary security policy will be described in Chapter III.



MMU_Image Array [Max_DBR_No MMU]
MMU Record [SDR Array [No_Seg_Desc_Reg
 Seg_Desc_Reg]

 Blks Used Word
 Max_Blks Word]

Seg_Desc_Reg Record [Base_Addr Address
 Limit Byte
 Attributes Byte]

Address Word

Figure 7. Memory Management Unit Image

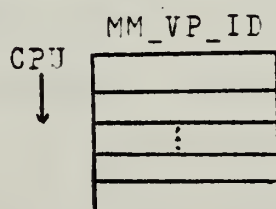


Figure 8. Memory Manager-CPU Table

III. SEGMENT MANAGEMENT IMPLEMENTATION

The implementation of segment management functions and a non-discretionary security policy is presented in this chapter. Paramount to this implementation were several key issues that affected the implementation. These issues are discussed first. The implementation is discussed in terms of the Segment Manager, Non-Discretionary Security (NDS), and Distributed Memory Manager modules.

A. IMPLEMENTATION ISSUES

Segment management for the SASS was provided through the implementation of the Segment Manager Module, the NDS Module, and the Distributed Memory Manager Module. Additionally, since a demonstration/testbed was integral to the testing and verification of the implementation, it was necessary to complete other supportive tasks. Reitz [9] provided a demonstration of the operation of the Inner Traffic Controller primitives SIGNAL and WAIT (for interprocess communication). Integral to this demonstration was the correct performance of the Inner Traffic Controller VP scheduling mechanism and a "stub" of the Traffic Controller and its process scheduling mechanism (the TC support and use of the mechanism of eventcounts and sequencers was not a part of the demonstration). The Segment management demonstration (hereafter referred to as

"Seg_Mgr.Demo") was "built on top of" Reitz' ITC synchronization primitive demonstration (hereafter referred to as "Sync. Demo"). Thus, an immediate issue was to resolve the feasibility of adding on to Sync.Demo and also to refine the present design of the Sync. Demo to facilitate its integration into the Seg_Mgr.Demo. One aspect of this effort was in resolving the problem of how to pass (i.e., in interprocess communication) a larger message.

1. Interprocess Messages

The Sync.Demo passed "word" (16 bit) messages. To provide the mechanism for the distributed memory manager to signal the memory manager process with a command function identification code and the arguments needed to perform that function (e.g., CREATE-ENTRY and its input arguments), a message size of at least eight words (16 bytes) was necessary. An obvious answer was to signal with an array of eight words as the message. PLZ/SYS, however, does not allow passing arrays in its procedure calls (a procedure call is analogous to a subroutine call). Another alternative was to signal with a pointer to the array of words, since PLZ/SYS does allow passing pointers in procedure calls (thus the message would be a pointer to the real message). This, however, would be invalid in the segmented implementation (on the 28000 segmented microprocessor) since identical segment numbers in different processes may not refer to identical segments. For example, a pointer in a process

(e.g., file management) points to an array (i.e., provides its address) by segment number and offset; passing this pointer to another process (e.g., memory manager) would provide this same segment number and offset which, of course, may be a different object in the second process's address space).

Another alternative considered was that of a shared "Mailbox" segment with an associated eventcount acted on by the Kernel Inner Traffic Controller primitives TICKET, ADVANCE and AWAIT. A design for using this concept in the supervisor ring is provided by Parks [7]. This alternative was not deeply considered since these primitives are not included in the current Inner Traffic Controller.

The method ultimately used to signal the new length messages is based on the fact that the ITC is in both the signalling and the receiving (memory manager) processes' address space. The message is loaded into an array in process #1 and a pointer to the array is passed in the call SIGNAL; the VPT, the ITC's database, is then updated by (using the pointer) putting the message into its MSG_Q section. The message is retrieved by process #2 by execution of Reitz' WAIT primitive with only one refinement. That refinement is for the "waiting" process to provide as an argument (in the WAIT primitive) a pointer to its own message array so that the message in the VPT can be copied to it.

This refinement provides for passing a long message essentially "by value" between processes.

2. Structures as Arguments

Another issue concerned the use of pointers in the implementation of segment management. This necessary "evil" is a result of the need to pass linguistically "complex" data types in procedure calls. Complex types refer to array and record structures in PLZ/SYS (as opposed to the "simple" types--byte, word, integer, short-integer, long, and pointer). In managing databases (e.g., KST, G_AST) which consist of arrays of records (which in turn contain records and/or arrays), it was frequently necessary to reference data as an array or record. Within a process, the use of pointers was not a problem (i.e., not a problem such as would be encountered in IPC passing of pointers).

3. Reentrant Code

The issue of code reentrancy was addressed at the assembly language level through the use of a stack segment and registers for storage of local variables. PLZ/SYS (high level language) does not address reentrant procedures and thus the segment management high level code is not automatically reentrant. The problem of reentrancy can be seen by looking at a shared procedure that is not reentrant; such a procedure has storage for its variables allocated statically in memory. Suppose a procedure (e.g., in the Kernel) can be activated by more than one process. While the

procedure is executing in one process, a process switch occurs (e.g., to wait for a disk transfer) and its execution is suspended. The second process is activated, and while it is running it invokes the procedure. While the procedure is executing for the second process it uses the same storage space for variables as it did when executing for the first process. Eventually, it relinquishes the processor. However, when the procedure resumes its execution for the first process, the variable values that were in use by it originally have been changed during its execution in the second process. Thus, incorrect results are now inevitable.

4. Process Structure of the Memory Manager

References to the "Memory Manager" in past works have generally meant the memory manager process (non-distributed kernel). This work references two distinct components of the "memory manager module". The Distributed Memory Manager is an interface provided to the Memory Manager Process. It is, in fact, distributed in the address space of each Supervisor process. In contrast, the Memory Manager Process clearly is not distributed and its address space is contained entirely in the Kernel.

5. Per-Process Known Segment Table

Another key issue was that of the per process Segment Manager database, the KST. Since each process has its own KST, it cannot be linked to the (shared) segment manager procedures. To implement the KST as a per process

database, it was convenient to establish, by convention, a KST segment number that is consistent from process to process. That segment in each process is the KST segment for that process. Implementation is then accomplished by using the segment number to construct a pointer to the base of the appropriate KST. It is then easy to calculate an appropriate offset to index any desired entry in the KST data.

6. DBR Handle

In Reitz's implementation of the multilevel scheduler and the IPC primitives, references to "DBR" (descriptor base register) are references to an address. That address value represents a pointer to an MMU_IMAGE record containing the list of descriptors for segments in the process address space. Gary and Moore [4] reference a "DBR_NO" that is essentially a handle used within the memory manager as an index within the MMU_IMAGE to a particular MMU record. The base address of the MMU record indexed by DER_NO is then equivalent to the concept of DBR value used in Reitz' work. The effect of this inconsistency on the segment management implementation was minor and will be further discussed later in this chapter.

B. SEGMENT MANAGER MODULE

The Segment Manager Module consists of six procedures representing the six extended instructions it provides. These are based on the design of Coleman [1]. Only calls

from external to the Kernel (via the Gate Keeper) may be made to the Segment Manager (per the loop-free structure of the SASS). The normal sequence of invocation of the Segment Manager functions to allow referencing a segment is: (1) `CREATE_SEGMENT`--allocate secondary storage for the segment and update the mentor segment's Alias Table, (2) `MAKE_KNOWN`--add the segment to the process address space (segment number is assigned), (3) `SWAP_IN`--move the segment from secondary storage into the process's main memory. The normal sequence of invocation to "undo" the above is: (1) `SWAP_OUT`--move the segment from main memory to secondary storage, (2) `TERMINATE`--remove the segment from the process's address space, (3) `DELETE_SEGMENT`--deallocate secondary storage and remove the appropriate entry from the alias table of its mentor segment. The six Supervisor entries into the Segment Manager (viz., the six extended instructions) will be discussed individually below. The PLZ/SYS and PLZ/ASM listings for the Segment Manager are in appendices A and B.

1. Create a Segment

The function that creates a segment (i.e., adds a new segment to the SASS) is `CREATE_SEGMENT`. This function validates the correctness of the Supervisor call by checking the parameters and making certain security checks. The distributed memory manager is then called to accomplish interprocess communication with the Memory Manager Process,

where segment creation is realized through secondary storage allocation and alias table updating.

CREATE_SEGMENT is passed as arguments: (1) Mentor_Seg_No--the segment number of the mentor segment of the segment to be created, (2) Entry_No--the desired entry number in the alias table of the mentor segment, (3) Class--the access class (label) of the segment to be created, and (4) Size--the desired size of the segment (in blocks of 256 bytes). The initial check is to verify that the desired size does not exceed the designed maximum segment size. If this check is satisfactory, a conversion of the Mentor_Seg_No to a KST index is necessary. This is because the Kernel segments use the first several segment numbers available but do not have entries in the KST. Thus if there were 10 Kernel segments and a system segment had segment number 15, then its index in the KST would actually be 5 (i.e., the Kernel segments would use numbers 0-9, and this segment would be the sixth segment in the KST and its index would be 5). A call is then made to the procedure ITC_GET_SEG_PTR with the constant KST_SEG_NO passed as a parameter. This procedure will return a pointer to the base of this process' KST. This pointer is then the basis for addressing entries in the KST. The next check is to see if the mentor segment is known (viz., is in the address space of the process, and thus, in the KST). The key to determining if any segment is known is the mentor segment

entry (M_SEG_No) for that segment in the KST. If not known, this entry in the segment's KST record will be filled with the constant NULL_SEG. The basis for checking to see if the segment's mentor segment is known is the aliasing scheme implication that a mentor segment must be known before a segment can be created. The process classification must next be obtained from the Traffic Controller. The process classification is checked to ensure that it is equal to the classification of the mentor segment since write access to its alias table is needed to create a segment. The NDS module's CLASS_EQ procedure is called and returns a code of true or false. The last check is the compatibility check to ensure that the classification of the segment to be created is greater than or equal to the classification of the mentor segment. This is accomplished by calling the NDS Module's CLASS_GE procedure which returns a code of true or false. If any of these checks are unsatisfactory, an appropriate error code is generated and the Segment Manager returns to its calling point. If all checks are satisfactory, then a pointer to the mentor segment's MM_Handle array is derived (HPTR). Note that in the current memory manager design [4] the actual MM_Handle contents are a Unique_ID (a long word, viz., two words concatenated), and an Index_No (index into the G_AST, a word); thus together these two fields are a total of three words. Since the Segment Manager does not interpret this handle, it is considered a three word array

at this level. For this reason, the entire uninterpreted MM_Handle array will be passed by passing its pointer. This pointer and Entry_No, Size, and Class are then passed in a call to the distributed memory manager procedure MM_CREATE_ENTRY. This procedure, in turn, performs IPC with the memory manager process where segment creation ultimately is accomplished. A success code is returned in an IPC message from the memory manager process via the distributed memory manager to the CREATE_SEGMENT procedure to indicate success or failure as appropriate. This success code is checked by the Segment Manager to ensure confinement would not be violated if it is returned to the calling process' supervisor domain. Only after the success code has been returned can the action of segment creation be considered complete. Segment creation does not imply the ability to reference that segment; MAKE_KNOWN will accomplish that.

2. Delete a Segment

The function that deletes a segment (i.e., deletes a segment from SASS) is DELETE_SEGMENT. Validation of parameters and security checks are performed here similar to (but fewer than) the CREATE_SEGMENT checks. The distributed memory manager is then called to cause IPC with the memory manager process, where segment deletion is realized through secondary storage deallocation and alias table entry deletions. DELETE_SEGMENT is passed as arguments: (1) Mentor_Seg_No and (2) Entry_No. Conversion of the

Mentor_Seg_No to a KST index is accomplished first. The pointer to the base of the KST is located and returned, as before. The mentor segment is checked to ensure it is known, again, by verifying that its own M_SEG_No (mentor segment number) entry in the KST is not the NULL_SEG. The process classification is obtained from the TC and checked (by a call to CLASS_EQ) to ensure it is equal to the mentor segment classification, since deleting an entry requires write access to the alias table. If all checks are satisfactory, then the mentor segment's MM_Handle pointer is derived. This pointer and the mentor segment alias table entry number are passed in a call to the distributed memory manager procedure MM_DELETE_ENTRY. It then performs IPC with the memory manager process where segment deletion is accomplished and a success code is returned as before.

3. Make a Segment Known

The function that makes a segment known (i.e., adds that segment to the process' address space by assigning a segment number, updating the KST, and causing the memory manager process to "activate" the segment (that is, add it to the AST)) is MAKE_KNOWN. Making a segment known is the way the Supervisor declares its intention to use a segment. MAKE_KNOWN is passed as arguments: (1) Mentor_Seg_No, (2) Entry_No, and (3) Access_Desired (e.g., write, read, or null). It returns (1) a success code, (2) the access allowed to the segment, and (3) the segment number. Conversion of

the mentor segment number to a KST index, finding the KST pointer, and verifying that the mentor segment is known occur as previously discussed.

There are three basic cases that may occur in MAKE_KNOWN: (1) the segment is already known (has an entry in the KST), (2) the segment is not known and there is a segment number available, or (3) the segment is not known and there is no segment number available.

A search is made of the KST using each record's (segment's) M_SEG_No (mentor segment number) and Entry_Number fields as the search key. If these two fields match the input values Mentor_Seg_No and Entry_No, then the record indexed is that of the desired segment; thus the segment to be made known is already known. In this case, all that need be done is to return the success code, segment number (converted from the index by adding to it the number of kernel segments), and the access allowed (equal to the Access_Mode entry in the KST for the already known segment).

During the search of the KST, the M_SEG_No field is also checked to see if it contains the NULL_SEG entry (this implies that the segment number associated with the record is "available"). The first time this is noted, the index is saved. Note the first available index is saved since it is desired to assign segment numbers at the "top" of the KST to keep it dense there. When the search does not find that the segment is already known, the index for the available

segment number is retrieved and converted to segment number by adding to it the number of kernel segments. If this index is the NULL_SEG entry, then there is no segment number available. In this event, the success code is set to NO_SEG_AVAIL, the segment number is assigned NULL_SEG, and access allowed is set to NULL_ACCESS (this is the third case mentioned). If the index is not equal to NULL_SEG and conversion to segment number has occurred then the Traffic Controller is called to provide the DBR_No (descriptor base register number) for the current process. The DBR_No is used by the memory manager process as an index in the MMU_Image and the local AST. The distributed memory manager procedure MM_Activate is called; it is passed the DBR number, the pointer to the mentor segment's MM_Handle entry, the mentor segment alias table Entry_No, and the segment number. MM_Activate performs the normal interface function (performs IPC with the memory manager process procedure that updates the local and global AST's) and also updates the KST entry for the new segment's MM_Handle entry (returned from the memory manager process). It also returns to the Segment Manager the success code, the segment classification, and the segment size from the memory manager process. If the success code is "succeeded" then the issue of access to be granted must be resolved. The process classification is obtained from the TC and passed with the segment classification to the NDS Module procedure CLASS_GE. If the

CONDITION_CODE returned is FALSE then access allowed is NULL_ACCESS, the segment number is NULL_SEG, and MM_DEACTIVATE is called to deactivate the segment. An appropriate error code is returned. If it is greater than or equal then the access allowed is assigned as follows: (1) the two classifications are compared again--this time to see if equal; (2) If they are equal, then the access allowed is either read or write per the access desired; (3) if they are not equal (i.e., the process class is greater than the segment class) then the access allowed is read. Finally the KST entries for that segment number (more accurately for its index in the KST) are filled with the appropriate information (e.g., IN_CORE is false, etc.). If the success code returned from the memory manager process via the distributed memory manager is not "succeeded", then the segment number is set to NULL_SEG and the access allowed is set to NULL_ACCESS.

4. Make a Segment Unknown (Terminate)

The function that makes a segment unknown (i.e., removes that segment from the process' address space--by updating the KST and causing the memory manager process to "deactivate" the segment) is TERMIMATE. It results in removal of the M_SEG_No (mentor segment number) entry from that segment's KST record. Terminate is passed the segment number of the segment to be terminated as an argument. It returns a success code. Conversion of the segment number to

a KST index, finding the KST pointer, and verifying that the segment is known occurs in the same manner as previously discussed. The next check is to verify that the segment is not still loaded in the process' virtual core (viz., it has been "swapped-out"). If not, an error code is returned and the user must cause the Segment Manager extended instruction SM_SWAP_OUT to be executed. The next check is to ensure that the user is not attempting to terminate a Kernel segment. The first several segment numbers in a process' address space will be used by Kernel procedures and data (though they will not be entries in the KST). Thus if there were 10 Kernel segments, then the segment number to be terminated must be greater than or equal to #10 (since the Kernel segments used #'s 0-9). Thus a check is made to ensure that the segment number is not less than the number of Kernel segments; otherwise an error code is returned. Next, the segment number is checked to ensure that it is not larger than the maximum segment number allowable (if so, an error code is returned). If all checks are satisfactory, then the segment's MM_Handle pointer and the process DBR_No are obtained (as discussed before) and passed in a call to the MM_Deactivate procedure. It calls the memory manager process procedure DEACTIVATE which removes or updates (as appropriate) the entries in the local and global AST's.

5. Swap a Segment In

The function that swaps a segment from secondary storage to main memory (global or local) is `SM_SWAP_IN`. It is passed the segment number of the segment to be swapped in as an argument and returns a success code. Conversion of the segment number to a KST index, finding the KST pointer, and verifying that the segment number is known are accomplished as previously discussed. If the check is satisfactory, then the segment's `MM_Handle` pointer and the process `DBR` number are obtained. They are passed with the segment's access mode (from the KST) as arguments in the call to `MM_SWAP_IN`. It performs normal interface (IPC) functions and returns a success code from the memory manager process' `SWAP_IN` procedure (where, if not already in core, allocation of main memory space and reading the segment into main memory occurs). If the success code is "succeeded" then the segment's `IN_CORE` entry in the KST is updated to show that the segment is in main memory for this process (i.e., the entry is now "true").

6. Swap a Segment Out

The function that swaps a segment from main memory to secondary storage is `SM_SWAP_OUT`. It is passed the segment number of the segment to be swapped out as an argument and returns a success code. The behavior of `SM_SWAP_OUT` is exactly analogous to that of `SM_SWAP_IN` except that the segment's KST `IN_CORE` entry is updated to

reflect that the segment has been removed from main memory for this process (i.e., the new entry is "false").

C. NON-DISCRETIONARY SECURITY MODULE

The Non-Discretionary Security Module implements the non-discretionary security policy for the SASS. The NDS module contains two procedures: CLASS_EQ and CLASS_GE; both compare two labels (classifications) and determine if their relationship meets that of the procedure's name (i.e., equal, or greater than or equal). Although the type of checks being made are, in fact, compatibility checks, Simple Security Condition checks, etc, the NDS Module does not recognize or need to recognize this. It simply uses an algorithm to determine if classification #1 = classification #2 or if classification #1 \geq classification #2, as appropriate. It then returns a condition code of true or false in accordance with the particular case. The earlier discussion of label comparison in accordance with a partially ordered lattice structure is relevant in discussing the NDS Module's algorithm. Consider the same "totally ordered" relationship $TS > S > C > U$ of levels and the "disjoint" relationship $Cy \mid N \mid Nu \mid \%$ of categories. Comparison of levels will be numerical comparisons while comparison of categories will use set theory comparison as a basis. If $TS=4$, $S=3$, $C=2$, $U=1$ are level numerical assignments, then the totally ordered relationship is

maintained (i.e., $TS > S > C > U$ is still true). Now consider the categories and make the following assignments: $Cy=1$, $N=2$, $Nu=4$, $\%=\emptyset$. Note that a classification may have only one level and one category set (the category set may contain several categories). Consider this example: $(TS, \{Cy, N\})$. The level is TS ($=4$). The category is the set $\{Cy, N\}$ and numerically is formed by performing a logical OR with the categories Cy and N . Sixteen bit representation of this is:

$Cy \text{ OR } N$

$(0000\ 0000\ 0000\ 0001) \text{ OR } (0000\ 0000\ 0000\ 0010)$

$= 0000\ 0000\ 0000\ 0011 = \{Cy, N\}$

If $(TS, \{Cy, N\})$ is considered label #1 and $(S, \{N\})$ as label #2 then a comparison of the two labels would be:

(1) Compare level #1 with level #2 -- $4 > 3$?

Clearly, the answer is yes.

(2) Compare category #1 with category #2 -- is

$(0000\ 0000\ 0000\ 0011)$ a superset of

$(0000\ 0000\ 0000\ 0010)$, or more clearly

is the latter a subset of the former?

The answer is yes, and one way to show that is true is by performing a logical OR of category #1 with category #2 and comparing the result to category #1. If the result of the OR operation equals category #1 then category #1 is a superset (not necessarily proper) of category #2. Since usage of the term subset is more frequent than that of superset, this relationship will typically be stated as

"category #2 is a subset of category #1. To illustrate the above:

{Cy,N} OR {N} :

(0000 0000 0000 0011) OR (0000 0000 0000 0010)

= 0000 0000 0000 0011 = category #1.

This means that, in this example, that category #2 is a subset (not necessarily proper) of category #1. Since level #1 > level #2 and category #2 subset category #1 then label #1 > label #2. Thus, a call to the CLASS_EQ procedure with these two labels as the input classifications would return a condition code of false while CLASS_GE would return true. The decision to have the classifications as long word (32 bits) supports the requirement of some DoD specifications for eight levels and sixteen categories. This module uses sixteen bits for the level and sixteen bits for the category. Appendices E and F are the PLZ/SYS and PLZ/ASM listings for the NDS Module.

1. Equal Classification Check

The CLASS_EQ procedure performs comparison of two classifications (labels) and returns a condition code of true if they are equal (an exact match of the two long words bit per bit) or false if they are not.

2. Greater or Equal Classification Check

The CLASS_GE procedure performs comparison of two classifications (labels) and returns a condition code true if classification #1 is greater than or equal to

classification #2 or a condition code of false otherwise. For classification #1 to be greater than or equal to classification #2, the following must be true: (1) level #1 \geq level #2 (determine this by simple numerical comparison of values) and (2) category #2 subset category #1 (determine this by performing a logical OR with the categories and comparing the result to category #1 -- if they are equal then category #2 is a subset of category #1).

Since PLZ/SYS allows passing only "simple" types in calls, the labels were passed as long words (as opposed to each being word arrays of length two). An access class label is never interpreted outside the NDS Module. However, within the NDS Module it is necessary to address the classification's components separately (viz., level and category). Thus, an "overlay" of the logical view of the classification was created. This overlay was a record of type ACCESS_CLASS and it consisted of two fields: level -- 16 bit integer and category -- 16 bit integer. A pointer type CPTR was declared to be of type pointer to ACCESS_CLASS. Two other pointers CLASS1_PTR and CLASS2_PTR were declared to be of type CPTR and were set equal to the base address of CLASS1 and CLASS2 respectively. This "overlay" of the record frame over the two classification labels passed as arguments allowed the desired component addressability.

Futhermore, the non-discretionary policy enforced by SASS can be changed from the current DoD policy to another lattice policy by changing (only) the NDS Module.

D. DISTRIBUTED MEMORY MANAGER MODULE

The Distributed Memory Manager Module performs as an interface between the Segment Manager and the Memory Manager Process. As its name implies, it is distributed in the kernel domain of each Supervisor process. The key role performed in this module is to arrange and perform interprocess communication between its process (actually the VP) and the memory manager process (VP). The module consists of eight procedures. Six of the procedures are called directly by Segment Manager procedures; they are MM_CREATE_ENTRY, MM_DELETE_ENTRY, MM_ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. The other two procedures are "service" procedures called by multiple procedures; they are: MM_GET_DBR_VALUE and PERFORM_IPC. The logic used in the first six procedures is somewhat uniform (except for MM_ACTIVATE). Thus, the general logic will be explained (with MM_CREATE_ENTRY as an example) and it should suffice as a description for all (except MM_ACTIVATE) procedures. The service procedures will be described separately.

1. Description of Procedures

Each procedure is invoked (and returns) on a one to one basis with a corresponding procedure in the Segment

Manager. For example, `CREATE_SEGMENT` invokes `MM_CREATE_ENTRY` which signals the `CREATE_ENTRY` procedure in the Memory Manager Process Module. Associated with each procedure is an IPC message "frame" to describe the unique format of the contents of the message to be signalled to the memory manager process. Similarly, there must be a message "frame" for return messages from the memory manager process; this frame is the same for all but the `MM_ACTIVATE` procedure. Consider the message frame for `MM_CREATE_ENTRY`; it consists of: (1) a code to describe which function is to be performed (e.g., `CREATE_CODE` indicates that the `CREATE_ENTRY` procedure is the intended recipient of the message), (2) `MM_Handle` (an array of three words), (3) `Entry_No`, (4) `Size`, and (5) `Class`. The message frame has a filler (in this case) of one byte to ensure that it is of length 16 bytes. The purpose of this frame is to provide an overlay onto the actual message array to be signalled and to facilitate loading the arguments into the message array. This is accomplished by having a pointer of the type that points to the frame but by converting its address so that it actually points to the base of the message array. Consider these lines of PLZ/SYS code:

```
CE_MSGPTR := CE_PTR COM_MSGPTR
```

```
CE_MSGPTR^.CREATE_CODE := CREATE_ENTRY_CODE
```

This code is putting a value into the structure pointed to by `CE_MSGPTR` at entry `CREATE_CODE`. The key point is that the

frame of that structure is, in fact, CREATE_MSG (as described before), but the physical location pointed to is the message array. This is assured by having the pointer CE_MSGPTR (which points to a structure of type CREATE_MSG) set equal to a pointer (COM_MSGPTR) to the actual message array (COM_MSGBUF). This is accomplished by the first line of code. The message array itself is never directly referenced, but rather the message array that is overlaid by the message frame is filled in the format of the CREATE_MSG frame. In this example, the first two bytes of the message array now contain the value of the constant CREATE_ENTR▼_CODE. The remainder of the message array is filled in the same manner (all procedures use the same notion of a frame, although the frames have different formats). The PERFORM_IPC (perform interprocess communication) procedure is called by all procedures at this point in their execution. The key is that the argument passed is the message array pointer not the pointer to the CREATE_MSG record (after all it is only an overlay frame -- linguistically, it is only a type and is never declared as a structure requiring memory storage allocation). When PERFORM_IPC returns, the message array contains a return message. This message consists of only a success code and filler space in all cases but MM_ACTIVATE. Interpretation of the return message is performed in the same manner as loading the message array. The retrieved success code is

returned to the calling Segment Manager procedure. For MM_ACTIVATE, the return message must be interpreted and values for success code, segment size, and segment classification retrieved and returned to the Segment Manager MAKE_KNOWN procedure. The value for the MM_Handle (called the G_AST_Handle by the memory manager process) must be retrieved and entered in the KST record for this segment.

2. Interprocess Communication

The final arrangements and actual performance of IPC is completed by the internal procedure PERFORM_IPC. By locating the identity of the current physical processor (CPU) and using that identity to index into the MM_CPU_TABLE, the VP_ID of the current memory manager is resolved, so that the memory manager process dedicated to this physical processor is signalled. The call to K_LOCK is, in fact, a disguised call to the SPIN_LOCK procedure (since K_LOCK calls SPIN_LOCK). K_LOCK represents an ultimate (as yet unimplemented) goal of a Kernel locking (wait-lock) system. In any event, the G_AST lock must be set prior to signalling the memory manager process. After SIGNAL has been called, a call is made to WAIT with the pointer to the message array as the argument. The synchronization cycle that results is: (1) PERFORM_IPC calls the ITC procedure SIGNAL with the memory manager VP_ID and message array pointer as arguments; PERFORM_IPC then calls WAIT with the message array as the argument, (2) SIGNAL causes the message

array to be copied into the message queue (in the VPT) of the appropriate VP_ID, (3) ultimately, the signalled VP is scheduled; it had previously called WAIT, passing a pointer to its own local message array; the action of WAIT is to copy the message from the VPT to the signalled process local message array; there it is interpreted by the memory manager process main procedure and the appropriate procedure is called for action (e.g., CREATE_ENTRY). (4) when action is completed the memory manager process fills its local message array with the appropriate return message and calls SIGNAL with a pointer to the message and the original signalling process's VP_ID as arguments, (5) SIGNAL causes the memory manager process' message to be copied into the VPT message queue for the appropriate VP_ID, (6) that VP is eventually scheduled and through the action of WAIT has the return message copied from its message queue in the VPT to its local message array; WAIT then returns to PERFORM_IPC. The G_AST lock is unlocked and PERFORM_IPC returns to the appropriate distributed memory manager procedure.

The last procedure in the distributed memory manager is MM_GET_DBR_VALUE. This procedure simply provides the service of translating a DBR_NO (DBR number) into its appropriate DBR address. It is called by the TC_GETWORK procedure to allow it to call the ITC procedure SWAP_VDBR (remember that presently the Inner Traffic Controller deals with the DBR as the address of the appropriate MMU record in

the MMU_IMAGE while the Traffic Controller uses DBR as a DBR number which indexes to the appropriate MMU record).

E. SUMMARY

The implementation of segment management functions and a non-discretionary security policy for the SASS has been presented in this chapter. The implementation of the Segment Manager Module, Non-Discretionary Security Module, and Distributed Memory Manager management demonstration was described.

Chapter IV will present the conclusions, lessons learned, and suggestions for future work derived from this thesis.

9200	EEEE	0000	DDDD	DDDD	0000	0040	CCCC	CCCC	*.....@.....*
9210	0300	0003	0001	0000	0001	0020	CCCC	CCCC	*.....*.....*
9220	0000	0000	0000	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*.....*
9230	0200	0003	0001	0000	0001	FFFF	CCCC	CCCC	*.....*.....*
9240	0000	0000	0000	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*.....*
9250	0000	0003	0001	0000	0002	0060	CCCC	CCCC	*.....<.....*
9260	0000	0000	0000	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*.....*
9270	0100	0003	0001	0000	0002	FFFF	CCCC	CCCC	*.....*.....*
9280	0000	0000	0000	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*.....*

I

Figure 9. Initialized Active Process Table

9000	0000	0060	0000	0000	CCCC	CCCC	CCCC	CCCC	* ..<.....*
9010	8000	0002	0001	0000	0000	0000	0040	FFFF	*@....*
9020	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	**
9030	8100	0001	0001	FFFF	FFFF	0000	0060	FFFF	*<....*
9040	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	**
9050	8100	0001	0001	FFFF	FFFF	0000	0020	FFFF	**
9060	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	**
9070	8100	0000	0000	0000	0000	0000	FFFF	FFFF	**
9080	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	**
9090	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	**
90A0	FFFF	0020	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	**
90B0	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	**
90C0	FFFF	0040	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*@.....*
90D0	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	**
90E0	FFFF	0060	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*<.....*
90F0	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	**
9100	FFFF	FFFF	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	**

Figure 10. Initialized Virtual Processor Table

[illegible]

Figure 13. Initialized Process Stack Segments


```

%ZLINK N=SM8 L=SM8.MAP MM.8 IDLE.8 IO.8 FM.8 TC.8 ITC.8 D MM.1 SM.1 ND SEC.8
PROC DATA = (IDLE DATA IO DATA FM DATA D MM DATA MM DATA)
KER PROC = (ITC INT PROC ITC GLB PROC TC GLB PROC D MM PROC SM PROC NDS PROC)
ZLINK N=SM8 L=SM8.MAP MM.8 IDLE.8 IO.8 FM.8 TC.8 ITC.8 D MM.1 SM.1 ND SEC.8 PROC
DATA = (IDLE DATA IO DATA FM DATA D MM DATA MM DATA) KER PROC = (ITC INT PROC I
TC GLB PROC TC GLB PROC D MM PROC SM PROC NDS PROC)
ZLINK 2.01
LINK COMPLETE
%IMAGER SM8 ($=5000 MM PROC $=5200 IDLE PROC $=5280 IO PROC $=5400 FM PROC $=560
0 KER PROC $=6200 PROC DATA $=8000 MMU DATA $=9000 ITC DATA $=9200 TC DATA) ^500
0 6500 O=SYNC.8
IMAGER SM8 ($=5000 MM PROC $=5200 IDLE PROC $=5280 IO PROC $=5400 FM PROC $=5600
KER PROC $=6200 PROC DATA $=8000 MMU DATA $=9000 ITC DATA $=9200 TC DATA) ^5000
6500 O=SYNC.8
IMAGER 2.0
1227 BYTES LOADED
%
```

Figure 14. Linker and Imager Command Lines

2
NMI
[LOAD SYNC.8
ENTRY POINT 5000
[LOAD DBR.8
ENTRY POINT 8000
[LOAD SYNC STACK.8
ENTRY POINT 7000
[LOAD SYNC DATA.8
ENTRY POINT 9000
[LOAD KST DATA.8
ENTRY POINT 9300
[R R15
R15 40A0 [71B4
RPC 075E [560E
RFC 5040 [5000
[

Figure 15. Load Command Lines and Register Initialization


```

IO: READ COMMAND
-----
FM: IO = SIGNALLER
FM: CALL KERNEL(CREATE)
-----
                                     KERNEL = SIGNALLER(FOR FM)
                                     MM: CREATE ENTRY
-----
FM: RETURN FROM KERNEL
-----
IO: READ COMMAND
-----
FM: IO = SIGNALLER
FM: CALL KERNEL(MAKE KNOWN)
-----
                                     KERNEL = SIGNALLER(FOR FM)
                                     MM: ACTIVATE
-----
FM: RETURN FROM KERNEL
-----
FM: CALL KERNEL(SWAP IN)
-----
NMI
[

```

Figure 16. Generated Output


```

-----
      KERNEL = SIGNALLER(FOR FM)
      MM: SWAP IN
-----
      FM: RETURN FROM KERNEL
-----
IO: READ COMMAND
-----
IO: CALL KERNEL(MAKE KNOWN)
-----
      KERNEL = SIGNALLER(FOR IO)
      MM: ACTIVATE
-----
IO: RETURN FROM KERNEL
-----
IO: CALL KERNEL(SWAP_IN)
-----
      KERNEL = SIGNALLER(FOR IO)
      MM: SWAP IN
-----
NMI
[

```

Figure 16. Generated Output (continued)


```

IG
IO: RETURN FROM KERNEL
-----
IO: READ COMMAND
-----
      FM: IO = SIGNALLER
      FM: CALL KERNEL(SWAP OUT)
-----
      KERNEL = SIGNALLER(FOR FM)
      MM: SWAP OUT
-----
      FM: RETURN FROM KERNEL
-----
      FM: CALL KERNEL(TERMINATE)
-----
      KERNEL = SIGNALLER(FOR FM)
      MM: DEACTIVATE
-----
      FM: RETURN FROM KERNEL
-----
IO: READ COMMAND
NMI
I

```

Figure 16. Generated Output (continued)


```

[ G
-----
IO: CALL KERNEL(SWAP OUT)
-----
                                KERNEL = SIGNALLER(FOR IO)
                                MM: SWAP OUT
-----
IO: RETURN FROM KERNEL
-----
IO: CALL KERNEL(TERMINATE)
-----
                                KERNEL = SIGNALLER(FOR IO)
                                MM: DEACTIVATE
-----
IO: RETURN FROM KERNEL
-----
IO: READ COMMAND
-----
                                FM: IO = SIGNALLER
                                FM: CALL KERNEL(DELETE)
-----
                                KERNEL = SIGNALLER(FOR FM)
-----
NMI
[

```

Figure 16. Generated Output (continued)

IG

```
MM: DELETE ENTRY
-----
FM: RETURN FROM KERNEL
-----
IO: READ COMMAND
-----
FM: IO = SIGNALLER
FM: CALL KERNEL(CREATE)
-----
KERNEL = SIGNALLER(FOR FM
```

NMI
I

Figure 16. Generated Output (continued)

IV. CONCLUSIONS AND FOLLOW ON WORK

The implementation of segment management for the security kernel of a secure archival storage system has been presented. The implementation was completed on Zilog's Z8002 sixteen bit nonsegmented microprocessor. Segmentation hardware (Zilog's Z8010 Memory Management Unit) was not available, therefore it was simulated in software as described by Reitz [9]. The loop free modular construction used in the implementation facilitates ease of expansion or modification.

A non-discretionary security policy was implemented using a partially ordered lattice structure as a basis. Enforcement was realized through an algorithm that compared two labels and determined if their relationship was equal to a desired relationship. Although the DoD security classification system was represented, any non-discretionary security policy that may be represented by a lattice structure may similarly be implemented. This implementation has shown that by having the non-discretionary security policy enforced in one module, changing to another policy requires changing only this one module.

Software engineering techniques used in previous work emphasized the advantages of working with code that is well structured, well documented, and well organized. Despite

being written in assembly language, Reitz' implementation of multiprogramming and process management proved to be consistent in style, clarity and documentation. This enhanced the construction of a segment management demonstration which was built onto his synchronization demonstration. Further, refinements made to his code (not necessitated by any failures of his code) were relatively easily accomplished.

While the segment management implementation appears to perform properly, it has not been subjected to a formal test plan. Such a test plan should be developed and implemented.

The Memory Manager Process has been designed but not implemented. Segment management implementation, provision for IPC using more practical size messages, and the detailed design of the memory manager by Moore and Gary [4], provide a sound foundation for memory manager implementation. A framework of the mainline code needed is provided in the memory manager module of the demonstration code in Appendix I. Prior to this implementation, formal testing of the segment management implementation herein and the monitor implemented by Reitz [9] should be completed.

APPENDIX A - SEGMENT MANAGER PLZ/SYS LISTINGS

SEGMENT_MANAGER

MODULE

CONSTANT

```

NULL_ACCESS      := 4
NULL_SEG         := -1
MAX_NO_KST_ENTRIES := ?? !TO BE DETERMINED!
MAX_SEG_SIZE     := ?? !TO BE DETERMINED!
MAX_SEG_NO       := ?? !TO BE DETERMINED!
KST_SEG_NO       := ?? !TO BE DETERMINED!
NR_OF_KSEGS      := ?? !TO BE DETERMINED!
FALSE            := 0
TRUE             := 1
READ             := 1
WRITE            := 2

```

! ****

```

SUCCESS_CODES    **** !
SUCCEEDED        := 2
MENTOR_SEG_NOT_KNOWN := 22
ACCESS_CLASS_NOT_EQ := 33
NOT_COMPATIBLE   := 24
SEGMENT_TOO_LARGE := 25
NO_SEG_AVAIL     := 27
SEGMENT_NOT_KNOWN := 28
SEGMENT_IN_CORE  := 29
KERNEL_SEGMENT   := 30
INVALID_SEGMENT_NO := 31
NO_ACCESS_PERMITTED := 32
LEAF_SEG_EXISTS  := 10
NO_LEAF_EXISTS   := 11
ALIAS_DOES_NOT_EXIST := 23
NO_CHILD_TO_DELETE := 20
G_AST_FULL       := 12
L_AST_FULL       := 13
LOCAL_MEMORY_FULL := 16
GLOBAL_MEMORY_FULL := 17
SEC_STOR_FULL    := 21
PROC_CLASS_NOT_GE_SEG_CLASS := 41

```

TYPE

```

H_ARRAY      ARRAY [ 3      WORD ]

KST_REC      RECORD [ MM_HANDLE      H_ARRAY
                      SIZE            WORD
                      ACCESS_MODE     BYTE
                      IN_CORE         BYTE
                      CLASS           LONG
                      M_SEG_NO        SHORT_INTEGER
                      ENTRY_NUMBER    SHORT_INTEGER ]

KST          ARRAY [ MAX_NO_KST_ENTRIES KST_REC ]

```


KSTPTR ^KST
ADDRESS WORD
SEG_ARRAY ARRAY [MAX_SEG_SIZE BYTE]

EXTERNAL

CLASS_EQ PROCEDURE
 RETURNS (CONDITION_CODE BYTE)

CLASS_GE PROCEDURE
 RETURNS (CONDITION_CODE BYTE)

MM_CREATE_ENTRY PROCEDURE
 RETURNS (SUCCESS_CODE BYTE)

MM_DELETE_ENTRY PROCEDURE
 RETURNS (SUCCESS_CODE BYTE)

MM_MAKE_KNOWN PROCEDURE
 RETURNS (SUCCESS_CODE BYTE
 CLASS LONG
 SIZE WORD)

MM_TERMINATE PROCEDURE
 RETURNS (SUCCESS_CODE BYTE)

MM_SWAP_IN PROCEDURE
 RETURNS (SUCCESS_CODE BYTE)

MM_SWAP_OUT PROCEDURE
 RETURNS (SUCCESS_CODE BYTE)

TC_GET_PROC_CLASS PROCEDURE
 RETURNS (PROC_CLASS LONG)

ITC_GET_SEG_PTR PROCEDURE
 RETURNS (SEGPTR ^SEG_ARRAY)

MONITOR PROCEDURE
!TO BE IMPLEMENTED AT ASSEMBLY LEVEL - SIMPLY WILL
CALL THE MONITOR AT ADDRESS %059A!

INTERNAL

HPTR ^H_ARRAY
KPTR KSTPTR

GLOBAL

```
!*****
*
* CREATE_SEGMENT PROCEDURE. INVOKED BY SUPERVISOR
* PROCEDURE VIA GATE KEEPER. ENSURES CALL IS VALID
* BY CHECKING TO SEE IF MENTOR SEGMENT KNOWN, IF
* SEGMENT SIZE IS TOO LARGE (DESIGNED MAX SIZE), IF
* ACCESS CLASSIFICATION ARE EQUAL, AND IF COMPATIBILITY
* REQUIREMENTS MET. IF ALL CHECKS ARE SATISFACTORY
* THEN MM_CREATE_ENTRY IS CALLED FOR MEMORY MANAGER
* TO TAKE ACTION.
*
*****!
```

```
CREATE_SEGMENT PROCEDURE ( MENTOR_SEG_NO      SHORT_INTEGER
                           ENTRY_NO           SHORT_INTEGER
                           CLASS              LONG
                           SIZE              WORD )
                           RETURNS ( SUCCESS_CODE  BYTE )
```

```
! **** NOTE: REENTRANT PROCEDURE **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !
LOCAL M_SEG_INDEX SHORT_INTEGER
ENTRY
  IF SIZE > MAX_SEG_SIZE THEN
    SUCCESS_CODE := SEGMENT_TOO_LARGE
  ELSE
    M_SEG_INDEX := MENTOR_SEG_NO - NR_OF_KSEGS
    KPTR := KSTPTR ITC_GET_SEG_PTR ( KST_SEG_NO )
    IF KPTR^[M_SEG_INDEX].M_SEG_NO = NULL_SEG THEN
      SUCCESS_CODE := MENTOR_SEG_NOT_KNOWN
    ELSE
      PROC_CLASS := TC_GET_PROC_CLASS
      CONDITION_CODE := CLASS_EQ ( PROC_CLASS,
                                   KPTR^[M_SEG_INDEX].CLASS )
      IF CONDITION_CODE = FALSE THEN
        SUCCESS_CODE := ACCESS_CLASS_NOT_EQ
      ELSE
        CONDITION_CODE := CLASS_GE ( CLASS,
                                     KPTR^[M_SEG_INDEX].CLASS )
        IF CONDITION_CODE = FALSE THEN
          SUCCESS_CODE := NOT_COMPATIBLE
        ELSE
          HPTR := #KPTR^[M_SEG_INDEX].MM_HANDLE
          SUCCESS_CODE := MM_CREATE_ENTRY ( HPTR,
                                             ENTRY_NO, SIZE, CLASS )
          CONFINEMENT_CHECK (SUCCESS_CODE)
        FI
      FI
    FI
  RETURN
END CREATE_SEGMENT
```



```

!*****
*
* DELETE_SEGMENT PROCEDURE. INVOKED BY SUPERVISOR
* PROCEDURE VIA THE GATE KEEPER. CHECKS TO SEE IF
* MENTOR SEGMENT KNOWN AND IF ACCESS CLASSIFICATION
* ARE EQUAL. THEN CALLS MM_DELETE_ENTRY FOR MEMORY
* MANAGER ACTION.
*
*****!

```

```

DELETE_SEGMENT PROCEDURE ( MENTOR_SEG_NO    SHORT_INTEGER
                           ENTRY_NO         SHORT_INTEGER )
      RETURNS ( SUCCESS_CODE    BYTE )

```

```

! **** NOTE: REENTRANT PROCEDURE **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL M_SEG_INDEX    SHORT_INTEGER
ENTRY
  M_SEG_INDEX := MENTOR_SEG_NO - NR_OF_KSEGS
  KPTR := KSTPTR ITC_GET_SEG_PTR ( KST_SEG_NO )
  IF KPTR^[M_SEG_INDEX].M_SEG_NO = NULL_SEG THEN
    SUCCESS_CODE := MENTOR_SEG_NOT_KNOWN
  ELSE
    PROC_CLASS := TC_GET_PROCC_CLASS
    CONDITION_CODE := CLASS_EQ ( PROC_CLASS,
                                   KPTR^[M_SEG_INDEX].CLASS )
    IF CONDITION_CODE = FALSE THEN
      SUCCESS_CODE := ACCESS_CLASS_NOT_EQ
    ELSE
      HPTR := #KPTR^[M_SEG_INDEX].MM_HANDLE
      SUCCESS_CODE := MM_DELETE_ENTRY ( HPTR, ENTRY_NO )
      CONFINEMENT_CHECK (SUCCESS_CODE)
    FI
  FI
  RETURN
END DELETE_SEGMENT

```



```

*****
*
* MAKE_KNOWN PROCEDURE. INVOKED BY SUPERVISOR
* PROCEDURE VIA GATE KEEPER. CHECKS TO SEE IF
* MENTOR SEGMENT KNOWN. SEARCHES KST TO SEE IF
* SEGMENT ALREADY KNOWN (WHILE ALSO NOTING THE
* FIRST AVAILABLE (UNUSED) SEG #). IF THE SEGMENT
* IS ALREADY KNOWN THEN FINISHED. IF NOT, THEN,
* USING THE AVAILABLE SEG #, CALL MM_ACTIVATE FOR
* MEMORY MANAGER ACTION. THEN CHECK TO SEE IF ANY
* ACCESS IS ALLOWED. IF YES THEN DETERMINE THE
* APPROPRIATE ACCESS ALLOWED AND UPDATE KST. IF
* NO, THEN RETURN NULL ACCESS.
*
*****!

```

```

MAKE_KNOWN PROCEDURE ( MENTOR_SEG_NO    SHORT_INTEGER
                      ENTRY_NO          SHORT_INTEGER
                      ACCESS_DESIRED    BYTE )
    RETURNS ( SEGMENT_NO    SHORT_INTEGER
              ACCESS_ALLOWED  BYTE
              SUCCESS_CODE    BYTE )

! **** NOTE:      REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL
    AVAIL_SEG SHORT_INTEGER
    INDEX      SHORT_INTEGER
    DER_NO     SHORT_INTEGER
    M_SEG_INDEX SHORT_INTEGER
ENTRY
    M_SEG_INDEX := MENTOR_SEG_NO - NR_OF_KSEGS
    KPTR := KSTPTR ITC_GET_SEG_PTR ( KST_SEG_NO )
    IF KPTR^[M_SEG_INDEX].M_SEG_NO = NULL_SEG THEN
        SUCCESS_CODE := MENTOR_SEG_NOT_KNOWN
        SEGMENT_NO := NULL_SEG
        ACCESS_ALLOWED := NULL_ACCESS
    ELSE
        PROC_CLASS := TC_GET_PROC_CLASS
        HPTR := #KPTR^[M_SEG_INDEX].MM_HANDLE
        INDEX := 0
        SEGMENT_NO := NULL_SEG
        AVAIL_SEG := NULL_SEG
        SEE_IF_KNOWN:
            DO
                IF KPTR^[INDEX].M_SEG_NO = MENTOR_SEG_NO
                ANDIF KPTR^[INDEX].ENTRY_NUMBER = ENTRY_NO THEN
                    !CASE: SEGMENT ALREADY KNOWN!
                    SUCCESS_CODE := SUCCEEDED
                    SEGMENT_NO := INDEX + NR_OF_KSEGS
                    ACCESS_ALLOWED := KPTR^[INDEX].ACCESS_MODE
                    EXIT SEE_IF_KNOWN

```



```

ELSE
  IF KPTR^[INDEX].M_SEG_NO = NULL_SEG
    ANDIF AVAIL_SEG = NULL_SEG THEN
      AVAIL_SEG := INDEX + NR_OF_KSEGS
    FI
    INDEX += 1
    IF INDEX > MAX_NO_KST_ENTRIES THEN EXIT FI
  FI
OD
! EXIT SEE_IF_KNOWN !
IF SEGMENT_NO = NULL_SEG
ANDIF AVAIL_SEG <> NULL_SEG THEN !CASE: SEGMENT NOT
      KNOWN AND SEG # IS AVAILAPLE!
  INDEX := AVAIL_SEG - NR_OF_KSEGS
  SEGMENT_NO := AVAIL_SEG
  DBR_NO := TC_GET_DBR_NO
  SUCCESS_CODE, KPTR^[INDEX].CLASS, KPTR^[INDEX].SIZE :=
    MM_ACTIVATE(DBR_NO, EPTR, ENTRY_NO, SEGMENT_NO)
  CONFINEMENT_CHECK (SUCCESS_CODE)
  IF SUCCESS_CODE = SUCCEEDED THEN
    CONDITION_CODE := CLASS_GE ( PROC_CLASS,
                                   KPTR^[INDEX].CLASS )
    IF CONDITION_CODE = FALSE THEN !NO ACCESS!
      ACCESS_ALLOWED := NULL_ACCESS
      SUCCESS_CODE := MM_DEACTIVATE (DBR_NO, EPTR)
      CONFINEMENT_CHECK (SUCCESS_CODE)
      SUCCESS_CODE := PROC_CLASS_NOT_GE_SEG_CLASS
      SEGMENT_NO := NULL_SEG
    ELSE
      CONDITION_CODE := CLASS_EQ ( PROC_CLASS,
                                    KPTR^[INDEX].CLASS )
      IF CONDITION_CODE = TRUE
      ANDIF ACCESS_DESIRED = WRITE THEN
        ACCESS_ALLOWED := WRITE
      ELSE
        ACCESS_ALLOWED := READ
      FI
      KPTR^[INDEX].IN_CORE := FALSE
      KPTR^[INDEX].M_SEG_NO := MENTOR_SEG_NO
      KPTR^[INDEX].ENTRY_NUMBER := ENTRY_NO
      KPTR^[INDEX].ACCESS_MODE := ACCESS_ALLOWED
    FI
  ELSE
    SEGMENT_NO := NULL_SEG
    ACCESS_ALLOWED := NULL_ACCESS
  FI
ELSE
  SUCCESS_CODE := NO_SEG_AVAIL
  SEGMENT_NO := NULL_SEG
  ACCESS_ALLOWED := NULL_ACCESS
FI
RETURN
END MAKE_KNOWN

```



```

!*****
*
*   TERMINATE PROCEDURE. INVOKED BY SUPERVISOR
*   PROCEDURE VIA GATE KEEPER. CHECKS TO SEE IF
*   SEGMENT IS KNOWN AND IF SEGMENT NUMBER IS
*   VALID. IF CHECKS ARE SATISFACTORY THEN MM_DEACTI-
*   VATE IS CALLED FOR MEMORY MANAGER ACTION.
*
*****!

```

```

TERMINATE PROCEDURE ( SEGMENT_NO  SHORT_INTEGER )
  RETURNS          ( SUCCESS_CODE BYTE )

```

```

! ****      NOTE: REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL      INDEX      SHORT_INTEGER
           DBR_NO      SHORT_INTEGER

```

```

ENTRY
  INDEX := SEGMENT_NO - NR_OF_KSEGS
  KPTR := KSTPTR ITC_GET_SEG_PTR , KST_SEG_NO ,
  IF KPTR^[INDEX].M_SEG_NO = NULL_SEG THEN
    SUCCESS_CODE := SEGMENT_NOT_KNOWN
  ELSE
    IF KPTR^[INDEX].IN_CORE = TRUE THEN
      SUCCESS_CODE := SEGMENT_IN_CORE
    ELSE
      IF SEGMENT_NO < NR_OF_KSEGS
        SUCCESS_CODE := KERNEL_SEGMENT
      ELSE
        IF SEGMENT_NO > MAX_SEG_NO THEN
          SUCCESS_CODE := INVALID_SEGMENT_NO
        ELSE
          HPTR := #KPTR^[INDEX].MM_HANDLE
          DBR_NO := TC_GET_DBR_NO
          SUCCESS_CODE := MM_DEACTIVATE (DBR_NO, HPTR)
          CONFINEMENT_CHECK (SUCCESS_CODE)
          IF SUCCESS_CODE = SUCCEEDED THEN
            KPTR^[INDEX].M_SEG_NO := NULL
          FI
        FI
      FI
    FI
  FI
  RETURN
END TERMINATE

```



```

!*****
*
* SM_SWAP_IN PROCEDURE. INVOKED BY SUPERVISOR
* PROCEDURE VIA THE GATE KEEPER. CHECKS TO SEE IF
* SEGMENT KNOWN; IF YES THEN CALLS MM_SWAP_IN FOR
* MEMORY MANAGER ACTION.
*
*****!

```

```

SM_SWAP_IN PROCEDURE ( SEGMENT_NO  SHORT_INTEGER )
    RETURNS          ( SUCCESS_CODE BYTE )

```

```

! ****      NOTE: REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL      INDEX      SHORT_INTEGER
           DBR_NO      SHORT_INTEGER

```

```

ENTRY
    INDEX := SEGMENT_NO - NE_OF_KSEGS
    KPTR := KSTPTR ITC_GET_SEG_PTR ( KST_SEG_NO )
    IF KPTR^[INDEX].M_SEG_NO = NULL_SEG THEN
        SUCCESS_CODE := SEGMENT_NOT_KNOWN
    ELSE
        IF KPTR^[INDEX].IN_CORE = TRUE THEN
            SUCCESS_CODE := SUCCEEDED
        ELSE
            HPTR := #KPTR^[INDEX].MM_HANDLE
            DBR_NO := TC_GET_DBR_NO
            SUCCESS_CODE := MM_SWAP_IN ( HPTR, DBR_NO,
                                         KPTR^[INDEX].ACCESS_MODE )
            CONFINEMENT_CHECK (SUCCESS_CODE)
            IF SUCCESS_CODE = SUCCEEDED THEN
                KPTR^[INDEX].IN_CORE := TRUE
            FI
        FI
    FI
    RETURN
END SM_SWAP_IN

```



```

!*****
*
* SM_SWAP_OUT PROCEDURE. INVOKED BY SUPERVISOR
* PROCEDURE VIA THE GATE KEEPER. CHECKS TO SEE IF
* SEGMENT KNOWN; IF YES THEN MM_SWAP_OUT IS CALLED
* FOR MEMORY MANAGER ACTION.
*
*****!

```

```

SM_SWAP_OUT PROCEDURE ( SEGMENT_NO SHORT_INTEGER )
    RETURNS ( SUCCESS_CODE BYTE )

```

```

! **** NOTE: REENTRANT PROCEDURE **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL INDEX SHORT_INTEGER
      DBR_NO SHORT_INTEGER

```

```

ENTRY
  INDEX := SEGMENT_NO - NR_OF_KSEGS
  KPTR := KSTPTR ITC_GET_SEG_PTR (KST_SEG_NO )
  IF KPTR^[INDEX].M_SEG_NO = NULL_SEG THEN
    SUCCESS_CODE := SEGMENT_NOT_KNOWN
  ELSE
    IF KPTR^[INDEX].IN_CORE = FALSE THEN
      SUCCESS_CODE := SUCCEEDED
    ELSE
      HPTR := #KPTR^[INDEX].MM_HANDLE
      DBR_NO := TC_GET_DBR_NO
      SUCCESS_CODE := MM_SWAP_OUT ( DBR_NO, HPTR )
      CONFINEMENT_CHECK (SUCCESS_CODE)
      IF SUCCESS_CODE = SUCCEEDED THEN
        KPTR^[INDEX].IN_CORE := FALSE
      FI
    FI
  FI
  RETURN
END SM_SWAP_OUT

```



```

!*****
*
* CONFINEMENT_CHECK PROCEDURE. SERVICE PPOCEDURE TO
* ENSURE NO SECURITY VIOLATION OCCURS WHEN INFO IS
* PASSED OUT OF THE KERNEL VIA THE SUCCESS_CODE.
* CALLS ASSEMBLY PROCEDURE - MONITOR WHICH IS TO
* CAUSE A JUMP TO THE MONITOR'S ADDRESS %259A.
*
*****!

```

CONFINEMENT_CHECK PROCEDURE (SUCCESS_CODE BYTE,

```

! ****      NOTE: REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

ENTRY

IF SUCCESS_CODE

```

CASE LEAF_SEG_EXISTS THEN
MONITOR !EXIT SYSTEM!
CASE NO_LEAF_EXISTS THEN
MONITOR !EXIT SYSTEM!
CASE ALIAS_DOES_NOT_EXIST THEN
MONITOR !EXIT SYSTEM!
CASE NO_CHILD_TO_DELETE THEN
MONITOR !EXIT SYSTEM!
CASE G_AST_FULL THEN
MONITOR !EXIT SYSTEM!
CASE L_AST_FULL THEN
MONITOR !EXIT SYSTEM!
CASE LOCAL_MEMORY_FULL THEN
MONITOR !EXIT SYSTEM!
CASE GLOBAL_MEMORY_FULL THEN
MONITOR !EXIT SYSTEM!
CASE SEC_STOR_FULL THEN
MONITOR !EXIT SYSTEM!

```

FI

RETURN

END CONFINEMENT_CHECK

END SEGMENT_MANAGER

APPENDIX B - SEGMENT MANAGER PLZ/ASM LISTINGS

2 !
3 !
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37

SEG_MGR MODULE

```

CONSTANT
NULL_SEG                      := -1
NULL_ACCESS                  := 4
MAX_SEG_NO                   := 64
MAX_NO_KST_ENTRIES           := 54
MAX_SEG_SIZE                 := 128
KST_SEG_NO                   := 2
NR_OF_KSEGS                  := 10
TRUE                          := 1
FALSE                         := 0
READ                          := 1
WRITE                         := 0
! **** SUCCESS_CODES **** !
SUCCEEDED                    := 2
MENTOR_SEG_NOT_KNOWN         := 22
ACCESS_CLASS_NOT_EQ         := 33
NOT_COMPATIBLE               := 24
SEGMENT_TOO_LARGE            := 25
NO_SEG_AVAIL                 := 27
SEGMENT_NOT_KNOWN            := 28
SEGMENT_IN_CORE              := 29
KERNEL_SEGMENT               := 30
INVALID_SEGMENT_NO           := 31
NO_ACCESS_PERMITTED         := 32
LEAF_SEG_EXISTS              := 10
NO_LEAF_EXISTS               := 11
ALIAS_DOES_NOT_EXIST         := 23
NO_CHILD_TO_DELETE           := 20
G_AST_FULL                   := 12
L_AST_FULL                   := 13
PROC_CLASS_NOT_OF_SEG_CLASS := 41

```

!PAGE


```

LOCAL MEMORY_FULL      := 16
GLOBAL_MEMORY_FULL     := 17
SEC_STOR_FULL          := 21
MONITOR                := %059A

```

```

TYPE      H_ARRAY      ARRAY [ 3      WORD ]

```

```

KST_REC   RECORD
[ MM_HANDLE
  SIZE      WORD
  ACCESS_MODE BYTE
  IN_CORE   BYTE
  CLASS     LONG
  M_SEG_NO  SHORT_INTEGER
  ENTRY_NUMBER SHORT_INTEGER]

```

```

ADDRESS      WORD

```

```

SEG_ARRAY  ARRAY [MAX_SEG_SIZE  BYTE]

```

```

INTERNAL

```

```

$SECTION KST_DCL
!NOTE: THIS SECTION IS AN OVERLAY/FRAME USED TO
DEFINE THE KST FORMAT. NO STORAGE IS ASSIGNED
RATHER THE KST IS STORED IN A SEPARATE
SEGMENT SET ASIDE FOR IT !

```

```

$ABS 0
KST   ARRAY [MAX_NO_KST_ENTRIES KST_REC]

```

2020

!PAGE

73	EXTERNAL
74	
75	CLASS_EQ PROCEDURE
76	
77	CLASS_GE PROCEDURE
78	
79	MM_CREATE_ENTRY PROCEDURE
80	
81	MM_DELETE_ENTRY PROCEDURE
82	
83	MM_ACTIVATE PROCEDURE
84	
85	MM_DEACTIVATE PROCEDURE
86	
87	MM_SWAP_IN PROCEDURE
88	
89	MM_SWAP_OUT PROCEDURE
90	
91	TC_GET_PROC_CLASS PROCEDURE
92	
93	ITC_GET_SEG_PTR PROCEDURE
94	
95	TC_GET_DBR_NO PROCEDURE
96	
97	
98	!PAGE


```

99      $SECTION SM_PROC
100      GLOBAL
101
102      CREATE_SEG      PROCEDURE
103
104      !*****!
105      ! CHECKS VALIDITY OF CREATE REQUEST AND !
106      ! CALLS MM_CREATE IF VALID. !
107      !*****!
108      ! REGISTER USE: !
109      ! PARAMETERS !
110      ! R1: MENTOR_SEG_NO(INPUT) !
111      ! R2: ENTRY_NO(INPUT) !
112      ! R3: SIZE(INPUT) !
113      ! R4: CLASS (INPUT) !
114      ! R0: SUCCESS_CODE (RETURNED) !
115      ! LOCAL USE !
116      ! R9: KST REC INDEX !
117      ! R6,R7_VARIOUS USES !
118      ! R13: ^KST !
119      !*****!
120
121      ENTRY
122      CP R3,#MAX_SEG_SIZE
123      IF GT THEN
124          LD R0,#SEGMENT_TOO_LARGE
125      ELSE
126          SUB R15,#10 !STACK AREA FOR INPUT REGS!
127          LDM R15,R1,#5
128          LD R1,#KST_SEG_NO
129          CALL ITC_GET_SEG_PTR !R1: KST_SEG_NO!
130                                     ! (RET:R0: ^KST)!
131          LD R13,R0 !KST BASE ADDRESS (IE ^KST)!
132          LDM R1,R15,#5 !RESTORE NEEDED REGS!
133      !PAGE

```


134	0026	A119	LD	R9,R1	!COPY OF MENTOR SEG_NO!
135	0028	0309	SUB	R9,#NR_OF_KSEGS	!CONVERT_MENTOR_SFG_NO
136		000A			KST_REC_INDEX!
137	002C	190E	MULT	RR8,#SIZEOF_KST_REC	!OFFSET TO KST_REC!
138	0030	819D	ADD	R13,R9	!ADD_OFFSET TO KST_REC!
139	0032	2106	LD	R6,#NULL_SEG	
140	0036	4ADE	CPB	RL6,ΔST.M_SEG_NO(R13)	
141	003A	5E0E	IF	EQ THEN	!MENTOR_SEG NOT KNOWN!
142	003E	2100	LD	R0,#MENTOR_SEG_NOT_KNOWN	
143	0042	5E0E	ELSE		
144	0046	93FD	PUSH	QR15,R13	
145	0048	5F00	CALL	TC_GET_PROC_CLASS	!(RR2:PROC_CLASS)!
146	004C	97FD	POP	R13,QR15	
147	004E	54D4	LDL	RR4,KST.CLASS(R13)	
148	0052	93FD	PUSH	QR15,R13	
149	0054	5F00	CALL	CLASS_EQ	!(RR2:PROC_CLASS)!
150		0000*			!(RR4:MENTOR_SEG_CLASS)!
151					!(R1:(RET)CONDITION_CODE)!
152	0058	97FD	POP	R13,QR15	
153	005A	A116	LD	R6,R1	
154	005C	1CF1	LDM	R1,QR15,#5	!RESTORE INPUT REGS!
155	0060	0B06	CP	R6,#FALSE	
156	0064	5E0E	IF	EQ THEN	
157	006E	2100	LD	R0,#ACCESS_CLASS_NOT_EQ	
158	006C	5F0E	ELSE		
159					
160	0070	93FD	PUSH	QR15,R13	!SAVE ^KST!
161	0072	9442	LDL	RR2,RR4	!CLASS!
162	0074	54D4	LDL	RR4,KST.CLASS(R13)	
163	0078	5F00	CALL	CLASS_GE	!(RR2:CLASS,!)
164		0000*			!(RR4:MENTOR_CLASS)!
165					!(RET:R1:COND_CODE)!
166	007C	97FD	POP	R13,QR15	!RESTORE PTR!
167	007E	0B01	CP	R1,#FALSE	
168	0082	1CF1	LDM	R1,QR15,#5	
169					!PAGE


```

190 DELETE_SEG
191 !*****PROCEDURE*****!
192 ! CHECKS VALIDITY OF DELETE REQUFST AND !
193 ! CALLS MM_DELETE IF VALID. !
194 !*****!
195 ! REGISTER USE: !
196 ! PARAMETERS !
197 ! R1:MENTOR_SEG_NO(INPUT) !
198 ! R2:ENTRY_NO(INPUT) !
199 ! R3:SUCCESS_CODE(RETURNED) !
200 ! LOCAL USE !
201 ! R6:VARIOUS LOCAL USES !
202 !*****!
203 !
204 !

```

```

ENTRY
205 PUSH @R15,R1 ISAVE NEEDED REGS!
206 PUSH @R15,R2
207 LD R1,#KST_SEG_NO
208 CALL ITC_GET_SEG_PTR IR1:KST_SEG_NO!
209 LD R13,R0 !~KST!
210 POP R2,@R15 !RESTORE INPUT REGS!
211 POP R1,@R15
212 SUP R1,#NR_OF_KSEGS !CONVERT MENTOR_SEG_NO TO
213 KST_REC_INDEX!
214 MULT RR0,#SIZEOF_KST_REC !OFFSET TO DESIRED REC!
215 ADD R13,R1 !ADD OFFSET TO KST BASE ADDRESS!
216 LD R6,#NULL_SEG
217 CPB RL6,ST_M_SEG_NO(R13)
218 IF EQ THEN !MENTOR_SEGMENT NOT KNOWN!
219 LD R0,#MENTOR_SEG_NOT_KNOWN
220 ELSE
221 PUSH @R15,R1 !SAVE NEEDED REGS!
222 PUSH @R15,R2
223 PUSH @R15,R13

```


!	00DA	5F00	0000*	225	CALL	TC_GET PROC CLASS
				226		!(RETURNS RR2:PROC_CLASS)!
	00DE	97FD		227	POP	R13,QR15
	00E0	54D4	000A	228	LDL	RR4,KST.CLASS(R13) IMENTOR SEG CLASS!
	00E4	93FD		229	PUSH	QR15,R13
	00E6	5F00	0000*	230	CALL	CLASS_EQ
				231		!(RR2:PROCESS CLASS)!
				232		!(RR4:MENTOR SEG CLASS)!
				233		!(R1:(RET) CONDITION_CODE)!
	00EA	A116		234	LD	R6,R1
	00FC	97FD		235	POP	R13,QR15
	00EE	97F2		236	POP	R2,QR15 !RESTORE NEEDED REGS!
	00F0	97F1		237	POP	R1,QR15
	00F2	0B06	0000	238	CP	R6,#FALSE
	00F6	5E0E	0102	239	IF	EQ THEN
	00FA	2100	0021	240	LD	R0,#ACCESS_CLASS_NOT_EQ
	00FE	5E0E	010E	241		
	0102	76D1	0000	242	LDA	R1,KST.MM_HANDLE(R13)
	0106	5F00	0000*	243	CALL	MM_DELETE_ENTRY
				244		!(R1:MM_HANDLE)!
				245		!(R2:ENTRY_NO)!
				246		!(R0:(RET)SUCCESS_CODE)!
	010A	5F00	041A	247	CALL	CONFINEMENT_CHECK
				248		!(R0:SUCCESS_CODE)!
				249	FI	
				250	RET	
	010E	9E08		251	END	DELETE_SEG
	0110			252	IPAGE	

!	0110		
253	0110	93F1	
254	0112	91F2	
255	0114	2101	0002
256	011E	5F00	0000*
257	011C	A10D	
258	011F	95F2	
259	0120	97F1	
260	0122	A115	
261	0124	0305	000A
262	0128	1904	0010
263	012C	815D	
264	012F	2104	FFFF
265	0132	4ADC	000E
266	0136	5F0E	014A
267	013A	2100	0016
268	013E	2101	FFFF
269			
270			
271			
272			
273			
274			
275			
276			
277			
278			
279			
280			
281			
282			
283			
284			
285			
286			
287			

```

MAKE KNOWN
!*****PROCEDURE*****!
! CHECKS VALIDITY OF MAKE KNOWN REQUEST AND !
! CALLS MM_ACTIVATE IF VALID. ASSIGNS SEG !
! NUMBER AND UPDATES KST.*****!
! REGISTER USE:
! PARAMETERS:
! R1:MENTOR_SEG_NO(INPUT)
! R2:ENTRY_NO(INPUT)
! R3:ACCESS_DESIRED(INPUT)
! R4:SUCCESS_CODE(RET)
! R1:SEGMENT_NO(RET)
! R2:ACCESS_ALLOWED(RET)
! LOCAL USE
! IDENTIFIED AT POINT OF USAGE*****!
ENTRY
  PUSH QR15,R1 !SAVE INPUT REGS!
  PUSHL QR15,RR2
  LD R1,#KST_SEG_NO
  CALL ITC_GET_SEG_PTR ! (R1:KST_SEG_NO,RET:R0:~KST)!
  LD R13,R0 !~KST!
  POPL RR2,QR15
  POP R1,QR15
  LD R5,R1 !COPY OF MENTOR_SEG_NO!
  SUE R5,#AR_OF_KSEGS !CONVERT TO INDEX!
  MULT RR4,#SIZEOF_KST_REC !KST OFFSET TO SEG REC!
  ADD R13,R5 !ADD OFFSET TO ~KST!
  LD R4,#NULL_SEG
  CPB R14,KST.M_SEG_NO(R13)
  IF EQ THEN
    LD R0,#MENTOR_SEG_NOT_KNOWN
    LD R1,#NULL_SEG
!PAGE

```


0142	2102	0004	LD	R2,#NULL_ACCESS	288
0146	5E08	02BA	ELSE		289
014A	2107	0000	LD	R7,#0 !KST INDEX!	290
014E	2108	FFFF	LD	R8,#NULL_SEG !AVAIL SEG INDEX!	291
0152	A109		LD	R9,R0 !KST!	292
0154	210A	FFFF	LD	R10,#NULL_SEG !SEG KNOWN INDICATOR!	293
			SEE_IF_KNOWN:		294
			DO		295
0158	4A99	000E	CPB	RL1,KST.M_SEG_NO(R9)	296
015C	5E0E	017C	IF EQ THEN		297
0160	4A9A	000F	CPB	RL2,KST.ENTRY_NUMBER(R9)	298
0164	5E0F	017C	IF EQ THEN !CASE: SEG KNOWN!		299
0168	2100	0002	LD	R0,#SUCCEEDED	300
016C	0107	000A	ADD	R7,#NR_OF_KSEGS	301
0170	A171		LD	R1,R7 !SEG#!	302
0172	609A	0008	LDB	RL2,KST.ACCESS_MODE(R9)	303
0176	A11A		LD	R10,R1 !SET SEG KNOWN INDICATOR!	304
017E	5E08	01A6	EXIT FROM SEE_IF_KNOWN		305
			FI		306
			FI		307
					308
017C	4A9C	000E	CPB	RL4,KST.M_SEG_NO(R9) !SEE IF SEG # AVAIL!	309
0180	5E0F	0192	IF EQ THEN		310
0184	0B28	FFFF	CP	R8,#NULL_SEG	311
0188	5E0E	0192	IF EQ THEN		312
018C	A178		LD	R8,R7 !SAVE FIRST AVAIL SEG INDEX!	313
018E	0108	000A	ADD	R8,#NR_OF_KSEGS !CONVERT TO SEG #!	314
			FI		315
			FI		316
0192	A970		INC	R7	317
0194	0109	0010	ADD	R9,#SIZEOF_KST_REC !INCREMENT ONE REC!	318
019E	0F07	0036	CP	R7,#MAX_NO_KST_ENTRIES	319
019C	5E02	01A4	IF GT THEN		320
01A0	5E08	01A6	EXIT FROM SEE_IF_KNOWN		321
					322 !PAGE

323	01A4	E8D9	FI	OD	IF	IF_KNOWN!
324	01A6	0B0A	FFFF	IF	R10,#NULL_SEG	IF_KNOWN!
325	01AA	5E0E	02BA	CP	THEN !SEG_KNOWN	INDICATOR NOT SET!
326	01AE	0R0E	FFFF	IF	CP	RS,#NULL_SEG
327	01B2	5E06	02AE	IF	NE THEN !CASE:SEG	UNKNOWN AND SEG# AVAIL!
328	01B6	91F0			PUSHL @R15,RR0	!KST AND MENTOR_SEG_NO!
329	01P8	91F2			PUSHL @R15,RR2	!ENTRY_NO &ACCESS_DESIRE!
330	01BA	93F8			PUSH @R15,R8	!SEG #!
331	01FC	93FD			PUSH @R15,R13	!MENTOR_SEG_REC_PTR!
332	01RE	5F00	0000*		CALL TC_GET_DER_NO	!(RET:R1:DER_NO)!
333	01C2	A11A			LD R10,R1	!DER_NO!
334	01C4	97FD			PCP R13,@R15	
335	01C6	97F8			POP RE,@R15	
336	01C8	95F2			PCPL RR2,@R15	
337	01CA	95FE			PCPL RR0,@R15	
338					!MUST REARRANGE REGS FOR PASSING AND	
339	01CC	A135			RETURN CONSISTENCY OF LOCATION!	
340	01CE	A123			LD R5,R3	!ACCESS_DESIRE!
341	01D0	76D2	0000		LD R3,R2	!ENTRY_NO!
342	01D4	A116			LDA R2,KST.MM_HANDLE(R13)	!MENTOR_HPTR!
343	01D6	A181			LD R6,R1	!MENTOR_SEG_NO!
344	01D8	A184			LD R1,RE	!SEGMENT_NO__(SAVE)!
345	01DA	A109			LD R4,R8	!SEGMENT_NO__(PASSING ARG)!
346	01DC	030F	0014		LD R9,R0	!KST!
347	01E0	1CF9	0109		SUB R15,#20	
348	01E4	A1A1			LDM @R15,R1,#10	!SAVE REGS 1-10!
349	01EC	5F00	0000*		LD R1,R10	!DER_NO PASSED IN R1!
350					CALL MM_ACTIVATE	
351	01FA	5F00	041A		!R1:DER_NO,R2:HPTR,R3:ENTRY_NO,	
352	01FE	942A			R4:SEGMENT_NO)!	
353					!(RET:R0:SUCCESS_CODE,RR2:CLASS,R4:SIZE)!	
354					CALL CONFINEMENT_CHECK ! (R0:SUCCESS_CODE)	
355					LDL RR10,RR2	!CLASS!
356						
357						
358						

01F0	A14C	359	LD	R12,R4	ISIZE!
01F2	1CF1	360	LJM	R1,QR15,#9	!RESTORE REGS 1-9!
01F6	A187	361	LD	R7,R8	!SEG #!
01F8	0307	362	SUB	R7,#NR	OF KSEGS
01FC	1906	363	MULT	RR6,#SIZEOF	KST_REC !OFFSET TO REC!
0200	A17D	364	LD	R13,R7	
0202	819D	365	ADD	R13,R9	!ADD ~KST TO OFFSET!
0204	5DDA	366	LDL	KST.CLASS(R13),RR10	!CLASS!
0208	6FDC	367	LD	KST.SIZE(R13),R12	!SIZE!
020C	0408	368	CPB	RL0,#SUCCEEDED	
0210	5E2E	369	IF	EQ	THEN
0214	93FD	370	PUSH	QR15,R13	
0216	5F00	371	CALL	TC_GET_PROC_CLASS	
		372	!(RET:RR2:PROC_CLASS)!		
		373	POP	R13,QR15	
021A	97FD	374	LDL	RR4,KST.CLASS(R13)	
021C	54D4	375	PUSH	QR15,R13	
0220	93FD	376	PUSHL	QR15,RR2	
0222	91F2	377	PUSHL	QR15,RR4	
0224	91F4	378	CALL	CLASS_GE	
0226	5F00	379	!(RR2:PROC_CLASS,RR4:SEG_CLASS,RET:		
		380	R1:CONDITION_CODE)!		
022A	95F4	381	POPL	RR4,QR15	
022C	95F2	382	POPL	RR2,QR15	
022E	97FD	383	POP	R13,QR15	
0230	0F01	384	CP	R1,#FALSE	
0234	5E0E	385	IF	EQ	THEN !NO ACCESS POSSIBLE--DEACT.!
0238	1CF1	386	LDN	R1,QR15,#10	
023C	A1A1	387	LD	R1,R10	!DPR NO!
023E	76D2	388	LDA	R2,KST.MM_HANDLE(R13)	!HPTR!
0242	5F00	389	CALL	MM_DEACTIVATE	!RET:R0:S_CODE!
0246	5F00	390	CALL	CONFINEMENT_CHECK	!RC:S_CODE!
024A	21F1	391	LD	R1,QR15	!SEG #!
024C	2102	392	LD	R2,#NULL	ACCESS
0250	2100	393	LD	R0,#PRCC_CLASS_NOT_GE_SEG_CLASS	
0254	5F08	394	ELSE		
0256	93FD	395	PUSH	QR15,R13	
		396			!PAGE

025A	5F00	0000*	397	CALL CLASS_EQ 1(RR2:PROC_CLASS,
025E	97FD		398	RR4:SEG CLASS,RET:R1:CONDITION_CODE)!
0260	A110		399	POP R13,GR15
0262	1CF1	0108	400	LD R0,R1 !CONDITION_CODE!
0266	0B00	0001	401	LDM R1,GR15,#9
026A	5E0E	0282	402	CP R0,#TRUE
026E	0B05	0000	403	IF EQ THEN
0272	5E0E	027C	404	CP R5,#WRITE
0276	CA00		405	IF EQ THEN
027E	5E0E	027E	406	LDB RL2,#WRITE
027C	CA01		407	ELSE
			408	LDB RL2,#READ
			409	FI
027F	5E0E	0284	410	ELSE
0282	CA01		411	LDB RL2,#READ
			412	FI
0284	4CD5	0009	413	LDB KST.IN_CORE(R13),#FALSE
028E	0000			
028A	6EDE	000E	414	LDB KST.M_SEG_NO(R13),RL6
028E	6EDE	000F	415	LDB KST.ENTRY_NUMBER(R13),RL3
0292	6EDA	0008	416	LDE KST.ACCESS_MOLF(R13),RL2
0296	2100	0002	417	LD R0,#SUCCEEDED !SUCCESS_CODE!
			418	FI
029A	5E08	02A6	419	ELSE
029E	2101	FFFF	420	LD R1,#NULL_SEG
02A2	2102	0004	421	LD R2,#NULL_ACCESS
			422	FI
02A6	010F	0014	423	ADD R10,#20
02AA	5E0E	02EA	424	ELSE
02AE	2100	001B	425	LD R0,#NO_SEG_AVAIL
02B2	2101	FFFF	426	LD R1,#NULL_SEG
02F6	2102	0004	427	LD R2,#NULL_ACCESS
			428	FI
			429	FI
			430	RFT
02FA	9E05		431	END MAKE_KNOWN
02BC			432	!PAGE
			433	

!

02FC

TERMINATE

PROCEDURE

434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468

028C A113
02FE 0303 000A
02C2 1902 0010
02C6 93F1
02C8 93F3
02CA 2101 0002
02CE 5F00 0000*

02D2 A10D
02D4 97F3
02D6 97F1
02D8 813D
02DA 2106
02DE 4ADE
02E2 5E0E
02E6 2100 001C
02EA 5E08 0338

```

!*****!
! CHECKS VALIDITY OF TERMINATE REQUEST !
! AND CALLS MM DEACTIVATE IF VALID !
!*****!
! REGISTER USE !
! PARAMETERS !
! R1:SEGMENT_NO(INPUT) !
! R0:SUCCESS_CODE(RETURNED) !
! LOCAL USE !
! R3:KST REC INDEX !
! R6:CONSTANT STORAGE !
! R13:~KST !
!*****!

```

ENTRY

```

LD R3,R1 ICOPIY OF SEG #!
SUB R3,#NR_OF_KSEGS !CONVERT SEG# TO KST INDEX!
MULT RR2,#SIZEOF KST_REC
PUSH @R15,R1
PUSH @R15,R3
LD R1,#KST SEG_NO
CALL ITC_GET_SEG_PTR ! (R1:KST_SEG_NO)!
! (RETURNS:R0:KST_SEG_PTR)!

LD R13,R0
POP R3,@R15
POP R1,@F15
ADD R13,R3 !ADD OFFSET TO ~KST!
LD R6,#NULL_SEG
CPB R16,KST.M_SEG_NO(R13)
IF EQ THEN
LD R0,#SEGMENT_NOT_KNOWN
ELSE

```

!PAGE

! 033A

SM_SWAP IN PROCEDURE

```
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
```

```
!*****!
! CHECKS VALIDITY OF SWAP IN REQUEST !
! AND CALLS MM_SWAP IN IF VALID !
!*****!
! REGISTER USE !
! PARAMETERS !
! R1:SEGMENT NO(INPUT) !
! R0:SUCCESS CODE(RETURNED) !
! LOCAL USE !
! R7:KST REC INDEX !
! R3:ACCESS MODE !
! R6:CONSTANT STORAGE !
! R13:~KST !
!*****!
```

ENTRY

```
LD R7,R1 !COPY OF SEG #!
SUB R7,#NR OF KSEGS !CONVERT SEG# TO KST INDEX!
MULT R6,#SIZE OF KST_REC !OFFSET TO KST_REC!
PUSH @R15,R1 !SAVE SEGMENT#!
PUSH @R15,R7
LD R1,#KST_SEG_NO
CALL ITC_GET_SEG_PTR !(R1:KST_SEG_NO)!
LD R13,R0 !~KST!
POP R7,@R15
POP R1,@R15 !RETRIEVE SEGMENT#!
ALD R13,R7 !ADD OFFSET TO KST BASE ADDR!
LD R6,#NULL_SEG
CPB R16,ΔST.M_SEG_NO(R13)
IF EQ THEN
LD R0,#SEGMENT NOT KNOWN
ELSE
```

!PAGE

036C	2106	0001	LD	R6,#TRUE	536
0370	4ADE	0009	CPR	RL6,KST.IN_CORE(R13)	537
0374	5E0F	0380	IF	EQ THEN	538
0378	2100	0002	LD	R0,#SUCCEEDED	539
037C	5E08	03AA	ELSE		540
0380	93FD		PUSH	QR15,R13 !SAVE KST REC ADDR!	541
0382	5F00	0000*	CALL	TC_GET_DBR_NO !R1:(RET)DBR_NO!	542
0386	97FD		POP	R13,QR15	543
0388	76D2	0000	LDA	R2,KST.MM_HANDLE(R13)	544
038C	60DB	0008	LDE	RL3,KST.ACCESS_MODE(R13)	545
0390	93FD		PUSH	QR15,R13 !SAVE SEG KST REC ADDR!	546
0392	5F00	0000*	CALL	MM_SWAP IN ! (R1:DPR_NO) !	547
				! (R2:MM_HANDLE)!	548
				! (R3:ACCESS_MODE)!	549
				! (R0:(RET)SUCCESS_CODE)!	550
0396	5F00	041A	CALL	CONFINEMENT_CHECK ! (R0:SUCCESS_CODE)!	551
039A	97FD		POP	R13,QR15	552
039C	0A08	0202	CPB	RL0,#SUCCEEDED	553
03A0	5E0E	03AA	IF	EQ THEN	554
03A4	4CD5	0009	LDB	KST.IN_CORE(R13),#TRUE	555
03A8	0101				556
			FI		557
			FI		558
03AA	9F08		RET		559
03AC			END SM_SWAP IN		560
			!PAGE		561

129

0466	5E08	04A6	642	CASE #L_AST_FULL THEN CALL MONITOR
046A	0E00	000D		
046E	5E0E	047A		
0472	5F00	059A	643	CASE #LOCAL_MEMORY_FULL THEN CALL MONITOR
0476	5E0E	04A6		
047A	0B00	0010		
047E	5E0E	048A	644	CASE #GLOBAL_MEMORY_FULL THEN CALL MONITOR
0482	5F00	059A		
0486	5E0E	04A6		
048A	0B00	0011		
048E	5F0F	049A		
0492	5F00	059A	645	CASE #SEC_STOR_FULL THEN CALL MONITOR
0496	5E08	04A6		
049A	0B00	0015		
049E	5E0E	04A6		
04A2	5F00	059A		
04A6	9E0E		646	FI
04A8			647	RET
			648	END CONFINEMENT_CHECK
			649	
			650	END SEG MGR
			651	
			652	!PAGE

APPENDIX C - DISTRIBUTED MEMORY MANAGER PLZ/SYS LISTINGS

DIST_MMGR MODULE

CONSTANT

```

CREATE_ENTRY_CODE      := 50
DELETE_ENTRY_CODE      := 51
ACTIVATE_SEG_CODE      := 52
DEACTIVATE_SEG_CODE    := 53
SWAP_IN_SEG_CODE       := 54
SWAP_OUT_SEG_CODE      := 55
NO_OF_PROCESSORS       := 1
MAX_NO_KST_ENTRIES     := ??
MAX_SEG_SIZE           := ??
MAX_DBR_NO             := 4
NR_OF_KSEGS            := ??
KST_SEG_NO             := ??

```

TYPE

```

H_ARRAY      ARRAY [3  WORD]

COM_MSG      ARRAY [16  BYTE]

CREATE_MSG    RECORD [CREATE_CODE  WORD
                      CE_MM_HANDLE H_ARRAY
                      CE_ENTRY_NO  SHORT_INTEGER
                      CE_FILLER    BYTE
                      CE_SIZE      WORD
                      CE_CLASS      LONG]

DELETE_MSG    RECORD [DELETE_CODE  WORD
                      DE_MM_HANDLE H_ARRAY
                      DE_ENTRY_NO  SHORT_INTEGER
                      DE_FILLER    ARRAY[7  BYTE]]

ACTIVATE_MSG  RECORD [ACTIVATE_CODE WORD
                      A_DBR_NO      SHORT_INTEGER
                      A_FILLER1     BYTE
                      A_MM_HANDLE    H_ARRAY
                      A_ENTRY_NO     SHORT_INTEGER
                      A_SEGMENT_NO   SHORT_INTEGER
                      A_FILLER2     LONG]

DEACTIVATE_MSG RECORD [DEACTIVATE_CODE WORD
                      D_DBR_NO      SHORT_INTEGER
                      D_FILLER1     BYTE
                      D_MM_HANDLE    H_ARRAY
                      D_FILLER2     ARRAY[3  WORD]]

```



```

SWAP_IN_MSG      RECORD [SWAP_IN_CODE  WORD
                        SI_MM_HANDLE  H_ARRAY
                        SI_DBR_NO     SHORT_INTEGER
                        SI_ACCESS_AUTH BYTE
                        SI_FILLER      ARRAY[3 WORD]]

SWAP_OUT_MSG     RECORD [SWAP_OUT_CODE WORD
                        SO_DBR_NO     SHORT_INTEGER
                        SO_FILLER1    BYTE
                        SO_MM_HANDLE  H_ARRAY
                        SO_FILLER2    ARRAY[3 WORD]]

R_SUC_CODE       RECORD [SUC_CODE      BYTE
                        SC_FILLER      ARRAY[15 BYTE]
                        ]

R_ACTIVATE_ARG   RECORD [R_SUC_CODE    BYTE
                        R_FILLER        BYTE
                        R_MM_HANDLE     H_ARRAY
                        R_CLASS         LONG
                        R_SIZE          WORD]

CE_PTR           ^CREATE_MSG

DE_PTR           ^DELETE_MSG

A_PTR            ^ACTIVATE_MSG

D_PTR            ^DEACTIVATE_MSG

SI_PTR           ^SWAP_IN_MSG

SO_PTR           ^SWAP_OUT_MSG

SC_PTR           ^R_SUC_CODE

ARG_PTR          ^R_ACTIVATE_ARG

KST_REC          RECORD [ MM_HANDLE      H_ARRAY
                        SIZE             WORD
                        ACCESS_MODE      BYTE
                        IN_CORE           BYTE
                        CLASS             LONG
                        M_SEG_NO          SHORT_INTEGER
                        ENTRY_NUMBER      SHORT_INTEGER ]

KST              ARRAY [ MAX_NO_KST_ENTRIES KST_REC ]

```


KSTPTR	^KST
ADDRESS	WORD
SEG_DESC_REG	RECORD [BASE_ADDR ADDRESS LIMIT BYTE ATTRIBUTE BYTE]
MMU	RECORD [SDR ARRAY [NO_SEG_DESC_REG SEG_DESC_REG] BLKS_USED WORD MAX_BLKS WORD]
MM_VP_ID	WORD
SEG_ARRAY	ARRAY [MAX_SEG_SIZE BYTE]
SEGPTR	^SEG_ARRAY

INTERNAL

MM_CPU_TABLE	ARRAY [NO_OF_PROCESSORS MM_VP_ID]
--------------	-----------------------------------

EXTERNAL

MMU_IMAGE	ARRAY [MAX_DBR_NO MMU]
G_AST_LOCK	BYTE
K_LOCK	PROCEDURE
K_UNLOCK	PROCEDURE
ITC_GET_CPU_NO	PROCEDURE
SIGNAL	PROCEDURE
WAIT	PROCEDURE


```

!*****
*
* MM_CREATE_ENTRY PROCEDURE. INVOKED BY SEGMENT
* MANAGER'S CREATE_SEGMENT PROCEDURE. PERFORMS TYPE
* CONVERSION OF POINTERS, LOADS MESSAGE ARRAY FOR
* IPC, AND PERFORMS IPC WITH MEMORY MANAGER PROCESS.
* RETURNS SUCCESS_CODE.
*
*****!

```

```

MM_CREATE_ENTRY PROCEDURE ( HPTR          ^H_ARRAY
                           ENTRY_NO      SHORT_INTEGER
                           SIZE          WORD
                           CLASS         LONG )
                           RETURNS ( SUCCESS_CODE BYTE )

```

```

! **** NOTE: REENTRANT PROCEDURE **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL CE_MSGPTR CE_PTR
      COM_MSGBUF COM_MSG
      COM_MSGPTR ^COM_MSG
      SUC_CODE_PTR SC_PTR

```

```

ENTRY
  COM_MSGPTR := #COM_MSGBUF
  ! TYPE CONVERT TO OVERLAY ARGUMENT LIST ONTO MSG LIST !
  CE_MSGPTR := CE_PTR COM_MSGPTR
  ! LOAD ARG LIST ONTO MSG ARRAY !
  CE_MSGPTR^.CREATE_CODE := CREATE_ENTRY_CODE
  CE_MSGPTR^.CE_MM_HANDLE[0] := HPTR^[0]
  CE_MSGPTR^.CE_MM_HANDLE[1] := HPTR^[1]
  CE_MSGPTR^.CE_MM_HANDLE[2] := HPTR^[2]
  CE_MSGPTR^.CE_ENTRY_NO := ENTRY_NO
  CE_MSGPTR^.CE_CLASS := CLASS
  CE_MSGPTR^.CE_SIZE := SIZE
  PERFORM_IPC (COM_MSGPTR)
  ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !
  SUC_CODE_PTR := SC_PTR COM_MSGPTR
  SUCCESS_CODE := SUC_CODE_PTR^.SUC_CODE
  RETURN
END MM_CREATE_ENTRY

```



```

!*****
*
* MM_DELETE_ENTRY PROCEDURE. INVOKED BY SEGMENT
* MANAGER'S DELETE_SEGMENT PROCEDURE. PERFORMS TYPE
* CONVERSION OF POINTERS, LOADS MESSAGE ARRAY FOR
* IPC, AND PERFORMS IPC WITH MEMORY MANAGER PROCESS.
* RETURNS SUCCESS_CODE.
*
*****!

```

```

MM_DELETE_ENTRY PROCEDURE ( HPTR           ^H_ARRAY
                           ENTRY_NO        SHORT_INTEGER)
      RETURNS ( SUCCESS_CODE  BYTE )

```

```

! **** NOTE:      REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL  DE_MSGPTR  DE_PTR
        COM_MSGBUF  COM_MSG
        COM_MSGPTR  ^COM_MSG
        SUC_CODE_PTR  SC_PTR

```

```

ENTRY
  COM_MSGPTR := #COM_MSGBUF
  ! TYPE CONVERSION TO OVERLAY ARG LIST ONTO MSG ARRAY !
  DE_MSGPTR := DE_PTR COM_MSGPTR
  ! LOAD ARG LIST ONTO MSG ARRAY !
  DE_MSGPTR^.DELETE_CODE := DELETE_ENTRY_CODE
  DE_MSGPTR^.DE_MM_HANDLE[0] := HPTR^[0]
  DE_MSGPTR^.DE_MM_HANDLE[1] := HPTR^[1]
  DE_MSGPTR^.DE_MM_HANDLE[2] := HPTR^[2]
  DE_MSGPTR^.DE_ENTRY_NO := ENTRY_NO
  PERFORM_IPC (COM_MSGPTR)
  ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO ARG LIST !
  SUC_CODE_PTR := SC_PTR COM_MSGPTR
  SUCCESS_CODE := SUC_CODE_PTR^.SUC_CODE
  RETURN
END MM_DELETE_ENTRY

```



```

!*****
*
* MM_ACTIVATE PROCEDURE. INVOKED BY SEGMENT MANA-
* GER'S MAKE_KNOWN PROCEDURE. PERFORMS TYPE CONVERSION
* OF POINTERS, LOADS MESSAGE ARRAY FOR IPC, AND UP-
* DATES MM_HANDLE ENTRY IN KST AFTER PERFORMING THE
* IPC. RETURNS SUCCESS_CODE, CLASS, AND SIZE.
*
*****!

```

```

MM_ACTIVATE PROCEDURE ( DBR_NO          SHORT_INTEGER
                        HPTR            ^H_ARRAY
                        ENTRY_NO        SHORT_INTEGER
                        SEGMENT_NO      SHORT_INTEGER )
    RETURNS ( SUCCESS_CODE BYTE
              CLASS        LONG
              SIZE         WORD )

```

```

! **** NOTE:      REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL  A_MSGPTR  A_PTR
        COM_MSGBUF  COM_MSG
        COM_MSGPTR ^COM_MSG
        RET_ARGPTR  ARGPTR
        KPTR        KSTPTR
        INDEX       SHORT_INTEGER

```

```

ENTRY
    COM_MSGPTR := #COM_MSGBUF
    ! TYPE CONVERT TO OVERLAY ARG LIST ONTO MSG ARRAY !
    A_MSGPTR := A_PTR COM_MSGPTR
    ! LOAD ARG LIST ONTO MSG ARRAY !
    A_MSGPTR^.ACTIVATE_CODE := ACTIVATE_SEG_CODE
    A_MSGPTR^.A_DBR_NO := DBR_NO
    A_MSGPTR^.A_MM_HANDLE[0] := HPTR^[0]
    A_MSGPTR^.A_MM_HANDLE[1] := HPTR^[1]
    A_MSGPTR^.A_MM_HANDLE[2] := HPTR^[2]
    A_MSGPTR^.A_ENTRY_NO := ENTRY_NO
    A_MSGPTR^.A_SEGMENT_NO := SEGMENT_NO
    PERFORM IPC (COM_MSGPTR)
    ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !
    RET_ARGPTR := ARG_PTR COM_MSGPTR
    SUCCESS_CODE := RET_ARGPTR^.R_SUC_CODE
    CLASS := RET_ARGPTR^.R_CLASS
    SIZE := RET_ARGPTR^.R_SIZE
    INDEX := SEGMENT_NO - NR_OF_KSEGS
    ! RETRIEVE HANDLE AND UPDATE KST !
    KPTR := KSTPTR MM_GET_KSEG_PTR ( SEG_NO )
    HPTR := #KPTR^[INDEX].MM_HANDLE
    HPTR^[0] := RET_ARGPTR^.R_MM_HANDLE[0]
    HPTR^[1] := RET_ARGPTR^.R_MM_HANDLE[1]
    HPTR^[2] := RET_ARGPTR^.R_MM_HANDLE[2]
    RETURN

```

```

END MM_ACTIVATE

```



```

!*****
*
* MM_DEACTIVATE PROCEDURE. INVOKED BY SEG MGR'S
* DEACTIVATE PROCEDURE. PERFORMS TYPE CONVERSION
* OF POINTERS, LOADS MESSAGE ARRAY FOR IPC, AND PER-
* FORMS IPC. RETURNS SUCCESS_CODE.
*
*****!

```

```

MM_DEACTIVATE    PROCEDURE ( DBR_NO      SHORT_INTEGER
                           HPTR        ^H_ARRAY )
                      RETURNS ( SUCCESS_CODE BYTE )

```

```

! **** NOTE:      REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL   D_MSGPTR    D_PTR
        COM_MSGBUF  COM_MSG
        COM_MSGPTR  ^COM_MSG
        SUC_CODE_PTR SC_PTR

```

```

ENTRY
  COM_MSGPTR := #COM_MSGBUF
  ! TYPE CONVERT TO OVERLAY ARG LIST ONTO MSG ARRAY !
  D_MSGPTR := D_PTR COM_MSGPTR
  ! LOAD ARG LIST ONTO MSG ARRAY !
  D_MSGPTR^.DEACTIVATE_CODE := DEACTIVATE_SEG_CODE
  D_MSGPTR^.D_DBR_NO := DBR_NO
  D_MSGPTR^.D_MM_HANDLE[0] := HPTR^[0]
  D_MSGPTR^.D_MM_HANDLE[1] := HPTR^[1]
  D_MSGPTR^.D_MM_HANDLE[2] := HPTR^[2]
  PERFORM_IPC (COM_MSGPTR)
  ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !
  SUC_CODE_PTR := SC_PTR COM_MSGPTR
  SUCCESS_CODE := SUC_CODE_PTR^.SUC_CODE
  RETURN
END MM_DEACTIVATE

```



```

!*****
*
* MM_SWAP_IN PROCEDURE. INVOKED BY SEGMENT MANAGER'S
* SM_SWAP_IN PROCEDURE. PERFORMS TYPE CONVERSION OF
* POINTERS, LOADS MESSAGE ARRAY FOR IPC, AND PERFORMS
* IPC. RETURNS SUCCESS_CODE.
*
*****!

```

```

MM_SWAP_IN  PROCEDURE  ( DBR_NO          SHORT_INTEGER
                        HPTR            ^H_ARRAY
                        ACCESS_MODE     BYTE )
                        RETURNS  ( SUCCESS_CODE  BYTE )

```

```

! **** NOTE:      REENTRANT  PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL  SI_MSGPTR  SI_PTR
        COM_MSGBUF  COM_MSG
        COM_MSGPTR  ^COM_MSG
        SUC_CODE_PTR  SC_PTR

```

ENTRY

```

COM_MSGPTR := #COM_MSGBUF
! TYPE CONVERT TO OVERLAY ARG LIST ONTO MSG ARRAY !
SI_MSGPTR := SI_PTR COM_MSGPTR
! LOAD ARG LIST ONTO MSG ARRAY !
SI_MSGPTR^.SWAP_IN_CODE := SWAP_IN_SEG_CODE
SI_MSGPTR^.SI_MM_HANDLE[0] := HPTR^[0]
SI_MSGPTR^.SI_MM_HANDLE[1] := HPTR^[1]
SI_MSGPTR^.SI_MM_HANDLE[2] := HPTR^[2]
SI_MSGPTR^.SI_DBR_NO := DBR_NO
SI_MSGPTR^.SI_ACCESS_AUTH := ACCESS_MODE
PERFORM_IPC (COM_MSGPTR)
! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !
SUC_CODE_PTR := SC_PTR COM_MSGPTR
SUCCESS_CODE := SUC_CODE_PTR^.SUC_CODE
RETURN

```

END MM_SWAP_IN


```

!*****
*
* MM_SWAP_OUT PROCEDURE. INVOKED BY SEGMENT MANAGER'S
* SM_SWAP_OUT PROCEDURE. PERFORMS TYPE CONVERSION OF
* POINTERS, LOADS MESSAGE ARRAY FOR IPC, AND PERFORMS
* IPC. RETURNS SUCCESS_CODE.
*
*****!

```

```

MM_SWAP_OUT    PROCEDURE    ( HPTR          ^H_ARRAY
                             ACCESS_MODE    BYTE )
                             RETURNS        ( SUCCESS_CODE  BYTE )

```

```

! **** NOTE:      REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL    SO_MSGPTR    SO_PTR
          COM_MSGBUF    COM_MSG
          COM_MSGPTR    ^COM_MSG
          SUC_CODE_PTR  SC_PTR

```

```

ENTRY
  COM_MSGPTR := #COM_MSGBUF
  ! TYPE CONVERT TO OVERLAY ARG LIST ONTO MSG ARRAY !
  SO_MSGPTR := SO_PTR COM_MSGPTR
  ! LOAD ARG LIST ONTO MSG ARRAY !
  SO_MSGPTR^.SWAP_OUT_CODE := SWAP_OUT_SEG_CODE
  SO_MSGPTR^.SO_MM_HANDLE[0] := HPTR^[0]
  SC_MSGPTR^.SO_MM_HANDLE[1] := EPTR^[1]
  SO_MSGPTR^.SO_MM_HANDLE[2] := HPTR^[2]
  SO_MSGPTR^.SO_DBR_NO := DBR_NO
  PERFORM_IPC (COM_MSGPTR)
  ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !
  SUC_CODE_PTR := SC_PTR COM_MSGPTR
  SUCCESS_CODE := SUC_CODE_PTR^.SUC_CODE
  RETURN
END MM_SWAP_OUT

```



```

!*****
*
* MM_GET_DBR_VALUE. SERVICE ROUTINE THAT RETRIEVES
* THE DBR "VALUE" (IE ADDRESS) BASED UPON A DBR_NO.
*
*****!

```

```

MM_GET_DBR_VALUE    PROCEDURE (DBR_NO    SHORT_INTEGER)
                    RETURNS      (DBR_VALUE ADDRESS);

```

```

! **** NOTE:      REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

ENTRY
    DBR_VALUE := #MMU_IMAGE [DBR_NO]
    RETURN
END MM_GET_DBR_VALUE

```

```

!*****
*
* PERFORM_IPC PROCEDURE. INVOKED BY DISTRIBUTED
* MEMORY MANAGER PROCEDURES. DETERMINES CURRENT
* MEMORY MANAGER'S VP_ID, LOCKS G_AST, PERFORMS
* IPC VIA SIGNAL AND WAIT PRIMITIVES, AND UNLOCKS
* THE G_AST.
*
*****!

```

```

PERFORM_IPC    PROCEDURE (COM_MSGPTR    ^COM_MSG)

```

```

! **** NOTE:      REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL    MMGR_VP_ID

```

```

ENTRY
    ! DETERMINE MMGR_VP_ID !
    CPU_NO := ITC_GET_CPU_NO
    MMGR_VP_ID := MM_CPU_TABLE [CPU_NO]
    K_LOCK (#G_AST_LOCK)
    ! IPC WITH MEMORY MANAGER PROCESS !
    SIGNAL ( COM_MSGPTR, MMGR_VP_ID )
    WAIT ( COM_MSGPTR )
    K_UNLOCK (#G_AST_LOCK )
    RETURN
END PERFORM_IPC

```

```

END DIST_MMGR

```



```

2 !
3 !
4 DIST_MM MODULE
5
6 CONSTANT
7
8 CREATE_ENTRY_CODE      := 50
9 DELETE_ENTRY_CODE      := 51
10 ACTIVATE_SEG_CODE     := 52
11 DEACTIVATE_SEG_CODE   := 53
12 SWAP_IN_SEG_CODE      := 54
13 SWAP_OUT_SEG_CODE     := 55
14 NO_OF_PROCESSORS      := 1
15 MAX_NO_KST_ENTRIES    := 54
16 MAX_SEG_SIZE         := 128
17 MAX_DBR_NO           := 4
18 KST_SEG_NO           := 2
19 NR_OF_KSEGS          := 10
20
21 TYPE
22
23 H_ARRAY                ARRAY [ 3 WORD ]
24
25 COM_MSG                ARRAY [16 BYTE]
26
27 KST_REC                RECORD [ MM_HANDLE      E_ARRAY
28                               SIZE             WORD
29                               ACCESS_MODE      BYTE
30                               IN_CORE         BYTE
31                               CLASS            LONG
32                               M_SEG_NO        SHORT_INTEGER
33                               ENTRY_NUMBER    SHORT_INTEGER ]
34
35 ADDRESS                WORD
36 ! PAGE

```



```

37 SEG_DESC_REG RECORD [BASE_ADDR ADDRESS
38 LIMIT BYTE
39 ATTRIBUTE BYTE]
40
41 MMU RECORD [SDR ARRAY [64
42 SEG_DESC_REG]]
43 !NOTE: NEXT TWO COMPONENTS
44 LEFT OUT FOR CONVENIENCE SINCE
45 ARE NOT USED FOR SEGMENT MGR
46 DEMO!
47 !BLKS_USED WORD
48 MAX_FLKS WORD]!
49
50 MM_VP_ID WORD
51
52 SEG_ARRAY ARRAY [MAX_SEG_SIZE BYTE]
53
54 INTERNAL
55
56 $SECTION D_MM_DATA
57 MM_CPU_TABLE_ARRAY [NO_OF_PROCESSORS MM_VP_ID] :=[0]
58
59 $SECTION MSG_FRAME_DCL
60 !NOTE: THESE RECORDS ARE "OVERLAYS" OR "FRAMES" USED
61 TO DEFINE MESSAGE FORMATS. NO MEMORY IS ALLOCATED FOR
62 THEM!
63 $ABS 0
64 CREATE_MSG RECORD [CREATE_CODE WORD
65 CE_MM_HANDLE E_ARRAY
66 CE_ENTRY_NO SHORT_INTEGER
67 CE_FILLER BYTE
68 CE_SIZE WORD
69 CE_CLASS LONG]
70
71 !PAGE

```


72	\$ABS 0				
73	DELETE_MSG	RECORD	[DELETE_CODE	WORD	
74			DE_MM_HANDLE	E_ARRAY	
75			DE_ENTRY_NO	SHORT_INTEGER	
76			DE_FILLER	ARRAY[7 BYTE]]	
77					
78	\$ABS 0				
79	ACTIVATE_MSG	RECORD	[ACTIVATE_CODE	WORD	
80			A_DBR_NO	SHORT_INTEGER	
81			A_FILLER1	BYTE	
82			A_MM_HANDLE	H_ARRAY	
83			A_ENTRY_NO	SHORT_INTEGER	
84			A_SEGMENT_NO	SHORT_INTEGER	
85			A_FILLER2	LONG]	
86					
87	\$ABS 0				
88	DEACTIVATE_MSG	RECORD	[DEACTIVATE_CODE	WORD	
89			D_DBR_NO	SHORT_INTEGER	
90			D_FILLER1	BYTE	
91			D_MM_HANDLE	H_ARRAY	
92			D_FILLER2	ARRAY[3 WORD]]	
93					
94	\$ABS 0				
95	SWAP_IN_MSG	RECORD	[SWAP_IN_CODE	WORD	
96			SI_MM_HANDLE	E_ARRAY	
97			SI_DBR_NO	SHORT_INTEGER	
98			SI_ACCESS_AUTH	BYTE	
99			SI_FILLER	ARRAY[3 WORD]]	
100	\$ABS 0				
101	SWAP_OUT_MSG	RECORD	[SWAP_OUT_CODE	WORD	
102			SO_DBR_NO	SHORT_INTEGER	
103			SO_FILLER1	BYTE	
104			SO_MM_HANDLE	H_ARRAY	
105			SO_FILLER2	ARRAY[3 WORD]]	
106					

!PAGE


```

107 $ABS 0
108 RET_SUC_CODE RECORD [SUC_CODE BYTE
109 SC_FILLER ARRAY[15 BYTE]
110 ]
111
112 $ABS 0
113 R_ACTIVATE_ARG RECORD [R_SUC_CODE BYTE
114 R_FILLER BYTE
115 R_MM_HANDLE H_ARRAY
116 R_CLASS LONG
117 R_SIZE WORD
118 R_FILLER1 WORD]
119
120
121 $ABS 0
122 KST ARRAY [ MAX_NO_KST_ENTRIES KST_REC ]
123
124
125 EXTERNAL
126 MMU_IMAGE ARRAY [MAX_DBR_NO MMU]
127
128 G_AST_LOCK WORD
129
130 K_LOCK PROCEDURE
131
132 K_UNLOCK PROCEDURE
133
134 ITC_GET_CPU_NO PROCEDURE
135
136 SIGNAL PROCEDURE
137
138 WAIT PROCEDURE
139
140 ITC_GET_SEG_PTR PROCEDURE
141 !PAGE

```



```

142 GLOBAL
143 $SECTION D_MM_PROC
144 MM_CREATE_ENTRY PROCEDURE
145 !*****!
146 ! INTERFACE BETWEEN SEG MGR !
147 ! (CREATE_SEG PROCEDURE) AND !
148 ! MMGR PROCESS (CREATE_ENTRY !
149 ! PROCEDURE). ARRANGES AND !
150 ! PERFORMS IPC. !
151 !*****!
152 ! REGISTER USE: !
153 ! PARAMETERS !
154 ! R0:SUCCESS CODE (RET) !
155 ! R1:HPTR (INPUT) !
156 ! R2:ENTRY_NO (INPUT) !
157 ! R3:SIZE (INPUT) !
158 ! R4:CLASS (INPUT) !
159 ! LOCAL USE !
160 ! R6:MM_HANDLE ARRAY ENTRY !
161 ! R8:~COM_MSGBUF !
162 ! R13:~COM_MSGBUF !
163 !*****!
164 ENTRY
165 SUP R15,#SIZEOF COM_MSG !USE STACK FOR MESSAGE!
166 LD R13,R15 !~COM_MSGBUF !
167
168 !FILL COM_MSGBUF (LOAD MESSAGE). CREATE MSG FRAME
169 !IS EASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
170 COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
171 ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
172
173 LD CREATE_MSG.CREATE_CODE(P13),#CREATE_ENTRY_CODE
174
175 LD R6,R1(#0) !INDEX TO MM_HANDLE ENTRY!
176 LD CREATE_MSG.CE_MM_HANDLE[0](R13),R6
177 LD R6,R1(#2)
177 !PAGE

```


0018	6FD6	0004	178	LD	CREATE_MSG.CE_MM_HANDLE[1](R13),R6
001C	3116	0004	179	LD	R6,R1(#4)
0020	6FD6	0006	180	LD	CREATE_MSG.CE_MM_HANDLE[2](R13),R6
0024	6FD2	0008	181	LD	CREATE_MSG.CE_ENTRY_NO(R13),R2
0028	5DD4	000C	182	LDL	CREATE_MSG.CE_CLASS(R13),RR4
002C	6FD3	000A	183	LD	CREATE_MSG.CE_SIZE(R13),R3
0030	A1DE		184	LD	R8,R13
0032	5F00	01A8	185	CALL	PERFORM_IPC !R8: ^COM MSGBUF!
			186	!RETRIEVE	SUCCESS_CODE FROM RETURNED MESSAGE!
0036	60DE	0000	187	LDB	RL0,RET_SUC_CODE.SUC_CODE(R13)
003A	010F	0010	188	ADD	R15,#SIZEOF_COM_MSG !RESTORE STACK STATE!
003E	9E08		189	RET	
0040			190	END MM_CREATE_ENTRY	
			191	!PAGE	


```

0040      MM_DELETE_ENTRY      PROCEDURE
192      !*****!
193      ! INTERFACE BETWEEN SEG MGR      !
194      ! (DELETE_SEG PROCEDURE) AND      !
195      ! MMGR (DELETE_ENTRY PROCEDURE). !
196      ! ARRANGES AND PERFORMS IPC.      !
197      !*****!
198      ! REGISTER USE:                    !
199      ! PARAMETERS                        !
200      ! R0: SUCCESS CODE (RET)           !
201      ! R1: HPTR (INPUT)                !
202      ! R2: ENTRY_NO (INPUT)            !
203      ! LOCAL USE                        !
204      ! R6: MM_HANDLE ARRAY ENTRY       !
205      ! R8: ^COM_MSGBUF                 !
206      ! R13: ^COM_MSGBUF                !
207      !*****!
208      !*****!
209      ENTRY
210      SUB  R15, #SIZEOF COM_MSG !USE STACK FOR MESSAGE!
211      LD   R13, R15 ! ^COM_MSGBUF !
212
213      ! FILL COM_MSGBUF (LOAD MESSAGE). DELETE_MSG FRAME
214      ! IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
215      ! COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
216      ! ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
217
218      LD   DELETE_MSG.DELETE_CODE(R13), #DELETE_ENTRY_CODE
219
220      LD   R6, R1(#0) ! INDEX TO MM_HANDLE ENTRY!
221      LD   DELETE_MSG.DE_MM_HANDLE[0](R13), R6
222      LD   R6, R1(#2)
223      LD   DELETE_MSG.DE_MM_HANDLE[1](R13), R6
224      LD   R6, R1(#4)
225      LD   DELETE_MSG.DE_MM_HANDLE[2](R13), R6
226      LD   DELETE_MSG.DE_ENTRY_NO(R13), R2
227      LD   R6, R13
228      ! PAGE

```


!	00CA 5F00	01A3'	228	CALL PERFORM IPC	IR8: ^COM MSGBUF!
			229	!RETRIEVE SUCCESS	COLE FROM RETURNED
	006E 60D8	0000	230	LDB	RL0,RET_SUC_CODE.SUC_CODE(R13)
	0072 010F	0010	231	ADD	R15,#SIZEOF_COM_MSG
	0076 9E08		232	RET	!RESTORE STACK STATE!
	0078		233		
			234	END MM_DELETE_ENTRY	
				!PAGE	


```

MM_ACTIVATE ***** PROCEDURE *****!
! INTERFACE BETWEEN SEG MGR !
! (MAKE_KNCWN PROCEDURE) AND !
! MMGR_ (ACTIVATE PROCEDURE). !
! ARRANGES AND PERFORMS IPC. !
! *****!
! REGISTER USE: !
! PARAMETERS !
! R0:SUCCESS CODE(RET) !
! R1:DBR_NO(INPUT) !
! R2:HPTR(INPUT) !
! R3:ENTRY_NO !
! R4:SEGMENT_NO !
! LOCAL USE !
! R8:~COM_MSGBUF !
! R13:~COM_MSGBUF !
! *****!

```

ENTRY

```

0078 030F 0010 SUB R15,#SIZEOF COM_MSG IUSE STACK FOR MESSAGE!
007C A1FD LD R13,R15 !~COM_MSGBUF !

007E 4DD5 0000 !FILL COM_MSGBUF (LOAD MESSAGE). ACTIVATE_MSG FRAME
0082 0034 IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
0084 6FD9 0002 COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
0088 3126 0000 ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
008C 6FD6 0004 LD ACTIVATE_MSG.ACTIVATE_CODE(R13),
0090 3126 0002 #ACTIVATE_SEG_CODE
0094 6FD6 0006 LD ACTIVATE_MSG.A_DBR_NO(R13),R1
0098 3126 0004 LD R6,R2(#0)
008C 6FD6 0004 LD ACTIVATE_MSG.A_MM_HANDLE[0](R13),R6
0090 3126 0002 LD R6,R2(#2)
0094 6FD6 0006 LD ACTIVATE_MSG.A_MM_HANDLE[1](R13),R6
0098 3126 0004 LD R6,R2(#4)

```

! PAGE

009C	6FD6	0008	271	LD	ACTIVATE_MSG.A_MM_HANDLE[2](R13),R6
00A0	6EDB	000A	272	LDB	ACTIVATE_MSG.A_ENTRY_NO(R13),R13
00A4	6EDC	000B	273	LDE	ACTIVATE_MSG.A_SEGMENT_NO(R13),R14
00A8	A1D8		274	LD	R8,R13
00AA	93F4		275	PUSH	QR15,R4 ISAVE COPY SEG #!
00AC	5F00	01A8	276	CALL	PERFCRM_IPC ! (RE: COM_MSGIUF!
			277		
			278		!UPDATE KST MM_HANDLE ENTRY!
00B0	2101	0002	279	LD	R1,#KST_SEG_NO
00B4	5F00	0000*	280	CALL	ITC_GET_SEG_PTR ! (R1:KST_SEG_NO,RET:P0:~KST)!
00F8	A10C		281	LD	R12,R0 !~KST!
00BA	97F4		282	PCP	R4,QR15 !SEG #!
00BC	A145		283	LD	R5,R4 !MOVE SEG # TO ALLOW MULT!
00FF	0305	000A	284	SUB	R5,#NR_OF_KSEGS !CONVERT TO INDEX!
00C2	1904	0010	285	MULT	RR4,#SIZEOF_KST_REC !OFFSET IN KST TO SIG REC!
00C6	815C		286	ADD	R12,R5 !ADD OFFSET TO ~KST!
00C8	61D6	0002	287	LD	R6,R_ACTIVATE_ARG.R_MM_HANDLE[0](R13)
00CC	6FC6	0000	288	LD	KST_MM_HANDLE[0](R12),R6
00D0	61D6	0004	289	LD	R6,R_ACTIVATE_ARG.R_MM_HANDLE[1](R13)
00D4	6FC6	0002	290	LD	KST_MM_HANDLE[1](R12),R6
00D8	61D6	0006	291	LD	R6,R_ACTIVATE_ARG.R_MM_HANDLE[2](R13)
00DC	6FC6	0004	292	LD	KST_MM_HANDLE[2](R12),R6
			293		!RETRIEVE OTHER RETURN ARGUMENTS!
00E0	60D8	0000	294	LDB	RL0,R_ACTIVATE_ARG.R_SUC_CODE(R13)
00E4	54D2	0008	295	LDL	RR2,R_ACTIVATE_ARG.R_CLASS(R13)
00E8	61D4	000C	296	LD	R4,R_ACTIVATE_ARG.R_SIZE(P13)
00EC	310F	0010	297	ADD	R15,#SIZEOF_COM_MSG !RESTORE STACK STATE!
00F0	9F38		298	.RET	
00F2			299		END MM_ACTIVATE
			300		!PAGE

!
00F2

```
MM_DEACTIVATE      PPOCEDURE
!*****!
! INTERFACE BETWEEN SEG MGR
! (TERMINATE PROCEDURE) AND
! MMGR (DEACTIVATE PROCEDURE).
! ARRANGES AND PERFORMS IPC.
!*****!
! REGISTER USE:
! PARAMETERS
! R0:SUCCESS CODE(RET)
! R1:DBR_NO(INPUT)
! R2:HPTR(INPUT)
! LOCAL USE
! R6:MM_HANDLE ARRAY ENTRY
! R8:~COM_MSGBUF
! R13:~COM_MSGBUF
!*****!
```

ENTRY

```
SUB R15,#SIZEOF COM_MSG !USE STACK FOR MESSAGE!
LD R13,R15 !~COM_MSGBUF !
```

```
!FILL COM_MSGBUF (LOAD MESSAGE). DEACTIVATE_MSG FRAME
IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
```

```
LD DEACTIVATE_MSG.DEACTIVATE_CODE(R13),
```

```
LIB DEACTIVATE_MSG.D_LPR_NO(R13),R11      #DEACTIVATE_SEG_CODE
LD R6,R2(#0) !INDEX TO MM_HANDLE ENTRY!
LD DEACTIVATE_MSG.D_MM_HANDLE[0](R13),R6
LD R6,R2(#2)
LD DEACTIVATE_MSG.D_MM_HANDLE[1](R13),R6
LD R6,R2(#4)
```

!PAGE

301			
302			
303			
304			
305			
306			
307			
308			
309			
310			
311			
312			
313			
314			
315			
316			
317			
318			
319			
320	00F2 030F	0010	
321	00F6 A1FD		
322			
323			
324			
325			
326			
327			
328	00F8 4DD5	0000	
	00FC 0035		
329			
330	00FE 6ED9	0002	
331	0102 3126	0000	
332	0106 6FD6	0004	
333	010A 3126	0002	
334	010E 6FD6	0006	
335	0112 3126	0004	
336			

!	0116	6FD6	0008	337	LD	DEACTIVATE_MSG.D_MM_HANDLE[2](R13),R6
	011A	A1DE		338	LD	R8,R13
	011C	5F00	01A8	339	CALL	PERFORM_IPC !R8: ^COM_MSGBUF!
				340		!RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
				341		
				342		
	0120	60DS	0000	343	LDB	RL0,RET_SUC_CODE.SUC_CODE(R13)
	0124	010F	0010	344	ADD	R15,#SIZEOF_COM_MSG !RESTORE STACK STATE!
	0128	9E08		345	RET	
	012A			346		END MM_DEACTIVATE
				347		!PAGE


```

012A
348 MM_SWAP_IN PROCEDURE
349 !*****!
350 ! INTERFACE BETWEEN SEG MGR (SM !
351 ! SWAP_IN PROCEDURE) AND MMGR !
352 ! (SWAP_IN PROCEDURE). ARRANGES !
353 ! AND PERFCRMS IPC. !
354 !*****!
355 ! REGISTER USE: !
356 ! PARAMETERS !
357 ! R0:SUCCESS CODE(RET) !
358 ! R1:DPR NO(INPUT) !
359 ! R2:HPTR(INPUT) !
360 ! R3:ACCESS_MODE(INPUT) !
361 ! LOCAL USE !
362 ! R6:MM_HANDLE_ARRAY ENTRY !
363 ! R8:~COM_MSGBUF !
364 ! R13:~COM_MSGBUF !
365 !*****!
366 ENTRY
367 SUB R15,#SIZEOF COM_MSG !USE STACK FOR MESSAGE!
368 LD R13,R15 !~COM_MSGBUF !
369
370 ! FILL COM_MSGBUF (LOAD MESSAGE). SWAP_IN_MSG FRAME
371 IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
372 COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
373 ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
374
375 LD SWAP_IN_MSG.SWAP_IN_CODE(R13),#SWAP_IN_SEG_CODE
376
377 LD R6,R2(#0) !INDEX TO MM_HANDLE ENTRY!
378 LD SWAP_IN_MSG.SI_MM_HANDLE[0](R13),R6
379 LD R6,R2(#2)
380 LD SWAP_IN_MSG.SI_MM_HANDLE[1](R13),R6
381 LD R6,R2(#4)
382 LD SWAP_IN_MSG.SI_MM_HANDLE[2](R13),R6
383 LD SWAP_IN_MSG.SI_DER_NO(R13),R1
384 ! PAGE

```



```

393 MM_SWAP_OUT PROCEDURE
394 !*****!
395 ! INTERFACE BETWEEN SEG_MGR (SM_!
396 ! SWAP_OUT PROCEDURE) AND MMGR_!
397 ! (SWAP_OUT PROCEDURE). ARRANGES!
398 ! AND PERFORMS IPC.
399 !*****!
400 ! REGISTER USE:
401 ! PARAMETERS
402 ! R0: SUCCESS CODE (RET)
403 ! R1: DBR_NO (INPUT)
404 ! R2: HPTR (INPUT)
405 ! LOCAL USE
406 ! R6: MM_HANDLE_ARRAY ENTRY
407 ! R8: COM_MSGBUF
408 ! R13: COM_MSGBUF
409 !*****!
410 ENTRY
411 SUB R15, #SIZEOF COM_MSG !USE STACK FOR MESSAGE!
412 LD R13, R15 ! COM_MSGBUF !
413
414 ! FILL COM_MSGBUF (LOAD MESSAGE). SWAP_OUT_MSG FRAME
415 ! IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
416 ! COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
417 ! ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
418
419 LD SWAP_OUT_MSG_SWAP_OUT_CODE(R13),
420
421 #SWAP_OUT_SEG_CODE
422 LD R6, R2(#2) ! INDEX TO MM_HANDLE_ENTRY!
423 LD SWAP_OUT_MSG_SO_MM_HANDLE[0](R13), R6
424 LD R6, R2(#2)
425 LD SWAP_OUT_MSG_SO_MM_HANDLE[1](R13), R6
426 LD R6, R2(#4)
427 LD SWAP_OUT_MSG_SO_MM_HANDLE[2](R13), R6
428 LD SWAP_OUT_MSG_SO_DBR_NO(R13), R11
429 !PAGE

```


!	018E	A1D8		LD	R8,R13	
	0190	5F00	01A8	CALL	PERFORM_IPC	!RE: ^COM_MSGBUF!
					!RETRIEVE_SUCCESS_CODE	FROM RETURNED MESSAGE!
	0194	60D8	0000	LDF	R10,RET_SUC_CODE	SUC_CODE(R13)
	0198	010F	0010	ADD	R15,#SIZEOF_COM_MSG	!RESTORE STACK STATE!
	019C	9E08		RET		
	019E				END MM_SWAP_OUT	
					IPAGE	


```

437 MM_GET_DBR_VALUE PROCEDURE
438 !*****!
439 ! RETRIEVES DBR VALUE (ADDRESS !
440 ! OF MMU REC POINTED TO BY DBR_ !
441 ! NO) !
442 !*****!
443 ! REGISTER USE: !
444 ! R1:DBR_NO (INPUT) !
445 ! R2:DBR_VALUE (RET) !
446 !*****!
447 !*****!
448 ENTRY
449 MULT RR0,#SIZEOF MMU
450 LDA R2,MMU_IMAGE(R1)
451 RET
452 END MM_GET_DBR_VALUE
453
454 !PAGE

```

```

019E 1900 0100
01A2 7612 0000*
01A6 9E08
01A8

```


! 01A8

```
455      PERFORM IPC      PROCEDURE
456      !*****!
457      ! SERVICE ROUTINE TO ARRANGE AND !
458      ! PFRFORM IPC WITH THE MEM MGR PROC !
459      !*****!
460      ! REGISTER USE: !
461      ! PARAMETERS !
462      ! R8: ^COM_MSG(INPUT) !
463      ! LOCAL USE !
464      ! R1,R2: WORK REGS !
465      ! R4: ^G_AST_LOCK !
466      ! R13: ^COM_MSGBUF !
467      !*****!
468
469      ENTRY
470      PUSH CR15,R13 !^COM_MSGBUF!
471      CALL ITC_GET_CPU_NO !RET-R1:CPU_NO!
472      LD R2,R1
473      LD R1,MM_CPU_TABLE(R2). IMM_VP_ID!
474      LDA R4,G_AST_LOCK
475      CALL K_LOCK
476      CALL SIGNAL IR1:MM_VP_ID,RE:^COM_MSGBUF!
477      POP R13,CR15
478      LD R8,R13 !^COM_MSGBUF!
479      PUSH CR15,R13
480      CALL WAIT IRS:^COM_MSGBUF!
481      LDA R4,G_AST_LOCK
482      CALL K_UNLOCK
483      POP R13,CR15
484      RET
485      END PERFORM_IPC
486
487      END DIST_MM
488
489      !PAGE
```

```
01A8 93FD
01AA 5F00 0000*
01AE A112
01B0 6121 0000*
01B4 7604 0002*
01B8 5F00 0000*
01BC 5F00 0000*
01C0 97FD
01C2 A1D8
01C4 93FD
01C6 5F00 0000*
01CA 7604 0000*
01CE 5F00 0000*
01D2 97FD
01D4 9F0E
01D6
```


APPENDIX E - NON-DISCRETIONARY SECURITY PLZ/SYS LISTINGS

NDS MODULE

CONSTANT

```
TRUE      := 1
FALSE     := 0
```

TYPE

```
ACCESS_CLASS      RECORD [ LEVEL  INTEGER
                           CAT    INTEGER ]
CPTR               ^ACCESS_CLASS
```

INTERNAL

```
CLASS1_PTR        CPTR
CLASS2_PTR        CPTR
CATS_REL          INTEGER
```

GLOBAL

```
!*****
*
*   CLASS_EQ PROCEDURE. INVOKED BY VARIOUS KERNEL
*   PROCEDURES. COMPARES TWO CLASSIFICATIONS (LABELS)
*   AND DETERMINES IF THEIR RELATIONSHIP IS EQUAL.
*   RETURNS A TRUE/FALSE CONDITION_CODE.
*
******!
```

```
CLASS_EQ PROCEDURE ( CLASS1      LONG
                     CLASS2      LONG )
                     RETURNS ( CONDITION_CODE BYTE )
```

ENTRY

```
IF CLASS1 = CLASS2 THEN
    CONDITION_CODE := TRUE
ELSE
    CONDITION_CODE := FALSE
FI
RETURN
END CLASS_EQ
```



```

!*****
*
*   CLASS_GE PROCEDURE. CALLED BY VARIOUS KERNEL
*   PROCEDURES. COMPARES TWO CLASSIFICATIONS (CLASS1
*   AND CLASS2) AND DETERMINES IF LABEL1 IS GREATER
*   THAN OR EQUAL TO CLASS2. RETURNS TRUE/FALSE CONDI-
*   TION CODE.
*
*****!

```

```

CLASS_GE PROCEDURE      ( CLASS1      LONG
                        CLASS2      LONG )
                        RETURNS      ( CONDITION_CODE BYTE )

```

```

ENTRY
! TYPE CONVERSION TO ALLOW OPERATIONS ON THE 16 MOST !
! AND 16 LEAST SIGNIFICANT BITS OF EACH CLASS. THE !
! 16 MSBITS ARE THE CLASS LEVEL AND THE 16 LSBITS !
! ARE THE CLASS CATEGORY (CAT). !

```

```

CLASS1_PTR := CPTR #CLASS1
CLASS2_PTR := CPTR #CLASS2
! COMPARE CATEGORIES (SET COMPARISON); SEE IF CAT2 !
! IS A SUBSET OF CAT1. !
CATS_REL := CLASS1_PTR^.CAT OR CLASS2_PTR^.CAT
IF CATS_REL = CLASS1_PTR^.CAT ! THEN CAT2 IS SUBSET !
ANDIF CLASS1_PTR^.LEVEL >= CLASS2_PTR^.LEVEL THEN
! LEVEL COMPARISON IS SIMPLE NUMERICAL COMPARISON !
    CONDITION_CODE := TRUE
ELSE
    CONDITION_CODE := FALSE
FI
RETURN
END CLASS_GE
END NDS

```



```

2 !
3 !
4 NDS MODULE
5 CONSTANT
6      :=1
7      :=0
8 INTERNAL
9 $SECTION ACC_CLASS_DCL !NOTE: IS AN OVERLAY, IE NO
10      ALLOCATION OF MEMORY!
11 $ABS 0
12 ACCESS_CLASS      RECORD [LEVEL  INTEGER
13      CAT           INTEGER]
14
15 GLOBAL
16 $SECTION NDS_PROC
17 CLASS_EQ PROCEDURE
18
19 !*****!
20 ! PASSED PARAMETERS !
21 ! RR2 = CLASS1      !
22 ! RR4 = CLASS2      !
23 ! RETURNED          !
24 ! R1 = CONDITION CODE !
25 !*****!
26
27 ENTRY
28     CPL      RR2,RR4
29     IF      EQ      THEN
30         LD      R1,#TRUE
31     ELSE
32         LD      R1,#FALSE
33     FI
34     RET
35 END CLASS_EQ
36 !PAGE

```


! 0014

CLASS_GE PROCEDURE

```
37
38
39 *****!
40 !! PASSED PARAMETERS !!
41 !! RR2 = CLASS1 !!
42 !! RR4 = CLASS2 !!
43 !! RETURNED PARAMETER !!
44 !! R1 = CONDITION_CODE !!
45 *****!
46
47 ENTRY
48 PUSHL @R15,RR2 !PUSH CLASS1 ON STACK--REFER BY ADDR!
49 LD R13,R15 ! CLASS1 ADDR !
50 PUSHL @R15,RR4
51 LD R14,R15 ! CLASS2 ADDR !
52 LD R7,R14(#ACCESS_CLASS.CAT) ! CAT2 IN R7 !
53 OR R7,ACCESS_CLASS.CAT(R13) !CAT1 OR CAT2, R7!
54 CP R7,ACCESS_CLASS.CAT(R13) !CAT1=(CAT1 OR CAT2)?!
55 IF EQ THEN
56 LD R6,ACCESS_CLASS.LEVEL(R13) !LEVEL1!
57 ! COMPARE LEVEL1 WITH IFVEL2 !
58 CP R6,ACCESS_CLASS.LEVEL(R14)
59 IF GE THEN ! LEVEL1 GE LEVEL2 !
60 LD R1,#TRUE
61 ELSE
62 LD R1,#FALSE
63 FI
64 ELSE
65 LD R1,#FALSE
66 FI
67 POPL RR4,@R15
68 PCPL RR2,@R15 !RESTORE STACK!
69 RET
70 END CLASS_GE
71 END NDS
72 !PAGE
```


APPENDIX G - SUMMARY OF REFINEMENTS

The following new procedures were added to the Inner Traffic Controller:

(1) ITC_GET_CPU_NO procedure. This procedure locates and returns the current CPU number (identification). It was used in segment management by the distributed memory manager to index into the MM_CPU_TABLE to find the memory manager process VP_ID for the current processor. This VP_ID was, in turn, used as an argument in the call to SIGNAL.

(2) ITC_GET_SEG_PTR procedure. This service procedure uses an input segment number to search the MMU_IMAGE to find the base address (pointer) for that segment. It was used in segment management to find the base address of the segment used in a process for its KST.

(3) K_LOCK and K_UNLOCK procedures. These procedures were implemented to indicate the intention to eventually have a kernel wait-lock system. K_LOCK simply calls SPIN_LOCK in its present design.

The following changes were made to the Inner Traffic Controller:

(1) The provision for a "jump table" was removed when a working version of the linker was introduced. This involved removing the constant TC_PREEMPT_HANDLER and adding an "external" declaration for the TC_PREEMPT_HANDLER Procedure.

(2) Minor changes were necessary in three procedures to modify the message size from one word to a sixteen byte array (minor changes were also needed in the declaration section). The procedures effected were: ENTER_MSG_LIST, GET_FIRST_MSG, and WAIT. The changes are documented in the code for each procedure.

The following procedures were added to the Traffic Controller:

(1) TC_GET_PROC_CLASS Procedure. This procedure locates and returns the current process's classification (i.e., it retrieves the SAC entry from the APT). It was used in segment management to retrieve the PROC_CLASS for the Segment Manager.

(3) TC_GET_DBR_NO Procedure. This procedure returns the current DBR_NO value from the APT. The Segment Manager used this procedure to obtain the DBR_NO to pass to the memory manager.

The version of the Traffic Controller shown in Appendix H is a "stub" of Reitz' [9] actual work. This stub contains the elements of the TC Module needed for proper operation of the segment management demonstration.

APPENDIX E - SEGMENT MANAGEMENT DEMONSTRATION

A. DESCRIPTION

The Seg_Mgr.Demo, as stated before, is built onto Reitz' Sync.Demo (which was designed for a different purpose than segment management, obviously). The functions illustrated by the present demonstration are: (1) virtual processor synchronization and (2) segment management function performance. The listings of the modules involved are in appendices A-F and I. It is suggested that the ASM versions be used as references; PIZ/SYS versions served as "pseudo-code" during detailed design, but are untested. The narrative discussion of the demonstration context and sequence is presented below. The output generated at each process entry point will identify the signaller in each case and the action the current process takes. The following actions are illustrated (viz., "simulated") in this demonstration. Note that this simulation uses the ITC SIGNAL/WAIT primitives instead of the TC ADVANCE/AWAIT primitives that are not yet implemented.

1. An I/O interrupt signals the IO process that a packet from the host is ready. The IO process is scheduled; it reads the packet (output: "IO: Receive Command") and signals the FM process (output: "IO: Signal FM (Create)"), passing the command (CREATE is

simulated). The IO process calls Wait, thus blocking itself while waiting for a signal from the FM process.

2. The FM process is scheduled (output: "IO=Signaller"); it interprets the command (simulated) as CREATE and thus calls CREATE_SEG in the Segment Manager (output: "FM: Call Kernel(Create)"). The normal input arguments for CREATE_SEG are passed in this call.

3. CREATE_SEG validates the CREATE request and then calls MM_CREATE_ENTRY in the Distributed Memory Manager.

4. MM_CREATE_ENTRY signals the memory manager process (MM process) and calls WAIT, thus blocking itself while waiting for a signal from the MM process.

5. The MM process is scheduled (output: "Kernel=Signaller (for FM)"); the mainline code interprets the function code and calls the CREATE_ENTRY procedure for action. When action is complete (output: "MM: CREATE_ENTRY"), the procedure returns to the mainline code. The MM process signals the return success code in a message to the FM process, then calls WAIT to wait for the next signal.

6. The FM process is scheduled (output: "Return from

Kernel"). The FM process then signals completion of CREATE to the IO process and calls WAIT (output: "FM: Signal IO").

7. The IO process is scheduled (output: "FM=Signaller"). It signals the FM process to cause a "read" to occur, i.e., the same pattern as in steps b. through e. occur for MAKE_KNOWN and SWAP_IN prior to the FM process signalling back to the IO process that the "read" was completed. This "read" is defined strictly for this test and is not equivalent to a typical Read_File packet.

8. The IO process will then be again scheduled and will perform the same functions as did the FM process (i.e., will call MAKE_KNOWN and SM_SWAP_IN sequentially) for the same segment.

9. The IO process will again be scheduled and will signal the FM process to perform the same sequence as described in g. for SM_SWAP_OUT and TERMINATE.

10. The IO process will again be scheduled and will repeat step h. with SM_SWAP_OUT and TERMINATE.

11. The IO process will again be scheduled and will cause the FM process to repeat steps b. through e. to delete (DELETE_SEG called) the segment.

12. The entire loop repeats forever.

E. INITIALIZATION

The description of the initialization of the databases is presented in figures containing the appropriate memory data. Reference to the previous descriptions of these databases and the type declarations will be useful. Figure 9 is the initialized Active Process Table. Figure 10 is the initialized Virtual Processor Table. Figure 11 is partial representation of the initialized KST for the FM process (9300) and the IO process (9700). Figure 12 is partial representation of the initialized MMU_IMAGE. Figure 13 is partial representation of the initialized process stack segments. Figure 14 is the corresponding link command line and response, and the Imager command line and response. Figure 15 is the load command lines and response, and the register initializations. Figure 16 is the output (as displayed on the CRT screen) generated by the demonstration.

APPENDIX 1 - DEMONSTRATION LISTINGS

INNER TRAFFIC CONTROL MODULE

```

10 1. GETWORK:
11 A. NORMAL ENTRY DOES NOT SAVE REGISTERS.
12 ( THIS IS A FUNCTION OF THE GATEKEEPER ).
13 C. R14 IS AN INPUT PARAMETER TO GETWORK THAT
14 SIMULATES INFO THAT WILL EVENTUALLY BE ON
15 THE MMU HARDWARE. THIS REGISTER MUST BE
16 ESTABLISHED AS A DBR BY ANY PROCEDURE
17 INVOKING GETWORK.
18 D. PREEMPT INTERRUPT ENTRY HANDLER, WHICH IS
19 CONTAINED IN GETWORK, DOES NOT USE THE
20 GATEKEEPER AND MUST PERFORM FUNCTIONS
21 NORMALLY ACCOMPLISHED BY IT
22 PRIOR TO NORMAL ENTRY AND EXIT.
23 ( SAVE/RESTORE: REGS, NSP; UNLOCK VPT,
24 TEST INT)
25
26 2. GENERAL:
27 A. ALL VIOLATIONS OF
28 VIRTUAL MACHINE INSTRUCTIONS ARE CONSIDERED
29 ERROR CONDITIONS AND WILL RETURN SYSTEM TO
30 MONITOR WITH ERROR CODE IN R0 AND PC IN R1.
31 B. ITC PROCEDURES CALLING GETWORK PASS DBR
32 (REGISTER R14) AS INPUT PARAMETER.
33 ( INCLUDES: SIGNAL, WAIT, SWAP_VDPR, AND
34 IDLE).

```

!PAGE


```

35 CONSTANT
36 ! ***** ERROR CODES ***** !
37 UNAUTH_LOCK := 0
38 MSG_LIST_EMPTY := 1
39 MSG_LIST_ERROR := 2
40 READY_LIST_EMPTY := 3
41 MSG_LIST_OVERFLOW := 4
42 SWAP_NOT_ALLOWED := 5
43 VP_INDEX_ERROR := 6
44
45
46 ! ***** SYSTEM PARAMETERS ***** !
47 NR_MMU_REG := 64 ! LONG WORDS!
48 NR_VP := 4
49 IDLE_VP := NR_VP-1
50 STACK_SEG := 1
51 STACK_SEG_SIZE := %100
52 ! * * * * * OFFSETS IN STACK SEG * * * !
53 STACK_BASE := STACK_SEG_SIZE-%40
54 STATUS_REG_BLOCK := STACK_SEG_SIZE-%40
55 F_C_W := STACK_SEG_SIZE-%20
56 PROCESS_ID := STACK_SEG_SIZE-%1E
57 N_S_P := STACK_SEG_SIZE-%1C
58 ON := %FFFF
59 OFF := 0
60 RUNNING := 0
61 READY := 1
62 WAITING := 2
63 NIL := %FFFF
64 INVALID := %FFFF
65 MONITOR := %A920 ! HBUG ENTRY !
66 ! PAGE

```



```

67 TYPE
68 MESSAGE
69 ADDRESS
70 VP_INDEX
71 MSG_INDEX
72
73 MMU_TABLE_RECORD [ BASE ADDRESS
74 ATTRIBUTES WORD
75 ]
76 MSG_TABLE_RECORD
77 [ MSG
78 SENDER
79 NEXT_MSG
80 FILLER
81 ]
82
83 VP_TABLE_RECORD
84 [ DBR
85 PRI
86 STATE
87 IDLE_FLAG
88 PREEMPT
89 PHYS_PROCESSOR WORD
90 NEXT_READY_VP VP_INDEX
91 MSG_LIST MSG_INDEX
92 FILLER_1 ARRAY [8, WORD]
93 ]
94
95 EXTERNAL
96 TC_PREEMPT_HANDLER PROCEDURE
97
98 !PAGE

```



```

99      INTERNAL
100      $SECTION ITC_DATA
101      VPT [ LOCK
102            RUNNING_LIST WORD
103            READY_LIST VP_INDEX
104            FREE_LIST VP_INDEX
105            FILLER_2 MSG_INDEX
106            VP ARRAY [4, WORD]
107            MSG_Q ARRAY [NR_VP, VP_TABLE]
108            ARRAY [NR_VP, MSG_TABLE]
109      ]
110 !PAGE

```

0000


```

111 $SECTION ITC_INT_PROC
112 GETWORK PROCEDURE
113 !*****!
114 ! SWAPS VIRTUAL PROCESSORS !
115 ! ON PHYSICAL PROCESSOR. !
116 !*****!
117 ! REGISTER USE: !
118 ! STATUS REGISTERS !
119 ! R0: INTERRUPT_RETURN_FLAG !
120 ! R14: DBR (SIMULATION) !
121 ! R15: STACK_POINTER !
122 ! LOCAL VARIABLES: !
123 ! R1: READY_VP (NEW) !
124 ! R2: CURRENT_VP (OLD) !
125 ! R3: FLAG_CONTROL_WORD !
126 ! R4: STACK_SEG_BASE_ADDR !
127 ! R5: STATUS_REG_BLOCK_ADDR !
128 ! R6: NORMAL_STACK_POINTER !
129 !*****!
130 ENTRY
131 ! TURN OFF PREEMPT_RETURN_FLAG !
132 LD R0, #OFF
133
134 ! GET STACK BASE !
135 LD R4, R14(#STACK_SEG*4)
136 IDA R5, R4(#STATUS_REG_BLOCK)
137
138 ! SKIP PREEMPT_HANDLER !
139 JR END_PFEEMPT_HANDLER
140
141 PREEMPT_ENTRY: ! GLOEAL LABEL !
142 ! * * PREEMPT_HANDLER * * !
143
144 !PAGE

```


!

145	000E 6102	0002
146	0012 612E	0010
147		
148		
149	0016 4D25	0014
150	001A 2001	
151		
152	001C 030F	0020
153	0020 1CF9	010F
154		
155		
156		
157	0024 7D67	
158	2026 93F6	
159		
160		
161		
162		
163		
164		
165		
166		
167		
168	0028 31E4	0004
169	002C 3445	0000
170		
171		
172	0030 1C51	0701
173	0034 93F7	
174	0036 93F8	
175		
176		

! PAGE

```
! SET DRR !
LD R2, VPT.RUNNING LIST
LD R14, VPT.VP.DBR(R2)

! PUT CURRENT PROCESS IN READY STATE !
LD VPT.VP.STATE(R2), #READY

! SAVE ALL REGISTERS !
SUB R15, #32
LDM QR15, R1, #16

! SAVE NORMAL STACK POINTER (NSP) !
LDCTL R6, NSP
PUSH QR15, R6

! SAVE LAST STATUS REGS !
!NOTE: SINCE PROCESSES CAN BE PREEMPTED ANYWHERE
IT IS NECESSARY TO HANDLE RECURSIVE CALLS
TO GETWORK. BY SAVING THE MOST RECENT SP
AND IRET_FLAGS (R15 & R0) ON THE STACK
THE CONTEXT OF THESE STATUS REGISTERS IS
MAINTAINED TO ANY DEPTH OF RECURSION. !

! GET STACK PASE !
LD R4, R14(#STACK_SEG*4)
LDA R5, R4(#STATUS_REG_BLOCK)

! SAVE LAST STATUS_REGS !
LDM R7, QR5, #2
PUSH QR15, R7
PUSH QR15, R8
```


0038	2100	FFFF	177	! SET INTERRUPT RETURN FLAG !
			178	LD R0, #ON
			179	! * * * * * !
			180	END_PREEMPT_HANDLER:
			181	
			182	! GET READY_VP LIST !
			183	LD R1, VPT.READY_LIST
			184	
			185	SELECT_VP:
			186	DO ! UNTIL ELGIBLE READY_VP FOUND !
			187	
			188	CP VPT.VP.IDLE_FLAG(R1), #ON
			189	IF EQ ! VP IS IDLE ! THEN
			190	CP VPT.VP.PREEMPT(R1), #ON
			191	IF EQ ! PREEMPT INTERRUPT IS ON ! THEN
			192	EXIT FROM SELECT_VP
			193	FI
			194	ELSE ! VP NOT IDLE !
			195	EXIT FROM SELECT_VP
			196	FI
			197	
			198	! GET NEXT READY_VP !
			199	LD R3, VPT.VP.NEXT_READY_VP(R1)
			200	LD R1, R3
			201	OD
			202	
			203	! NOTE: THE READY LIST WILL NEVER BE EMPTY SINCE
			204	THE IDLE VP, WHICH IS THE LOWEST PRI VP,
			205	WILL NEVER BE REMOVED FROM THE LIST.
			206	IT WILL RUN ONLY IF ALL OTHER READY_VP'S ARE
			207	IDLING OR IF THERE ARE NO OTHER VP'S ON
			208	THE READY_LIST. ONCE SCHEDULED, IT
			209	WILL RUN UNTIL RECEIVING A HDWE INTERRUPT. !
			210	
			211	! PAGE
			212	


```

211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
006E 1C59 0F01
006C 7D32
006E 3343 00E0
0072 4D15 0014
0076 0000
0078 6F01 0002
007C 611E 0010
0080 31E4 0004
0084 3445 00C0
0088 1C51 0F01
00EC 3143 00E0
0090 7D3A
0092 0F00 FFFF
009C 5E0E 00BC
009A 4D0E 0000
! PAGE

```

```

! NOTE: R14 IS USED AS DBR HERE. WHEN MMU
IS AVAILABLE THIS SERIES OF SAVE AND LOAD
INSTRUCTIONS WILL BE REPLACED BY SPECIAL I/O
INSTRUCTIONS TO THE MMU. !
! * * SAVE SP AND INTERRUPT RETURN FLAG * * !
LDM @R5, R15, #2

! * * SAVE FCW * * !
LDCTL R3, FCW
LD R4(#F_C_W), R3

! PLACE NEW VP IN RUNNING STATE !
LD VPT.VP.STATE(R1), #RUNNING

LD VPT.RUNNING_LIST, R1

! * * SWAP DBR * * !
LD R14, VPT.VP.DBR(R1)

! LOAD NEW_VP SP & INTERRUPT RET FLAG !
LD R4, R14(#STACK_SEG*4)
LDA R5, R4(#STATUS_REG_BLOCK)
LDM R15, @R5, #2

! * * LOAD NEW FCW * * !
LD R3, R4(#F_C_W)
LDCTL FCW, R3

! TEST FOR HARDWARE INTERRUPT !
CP R0, #ON
IF EQ ! PREEMPT RETURN ! THEN
! HARDWARE PREEMPT INTERRUPT RETURN !

```

```

! UNLOCK VPT !
CLR VPT.LOCK

```


246		
247		
248		
249		
250		
251	009E 5F00	0106'
252		
253		
254	00A2 97F8	
255	00A4 97F7	
256	00A6 1C59	0701
257		
258		
259	00AA 97F6	
260	00AC 7D6F	
261		
262	00AF 1CF1	010F
263	00B2 010F	0020
264		
265		
266	00B6 7B00	
267		
268	00FE 5E08	00EF'
269	00BC 9E28	
270		
271	00BF	
272		!PAGE

```

! TEST FOR PREEMPT !
! NOTE: SINCE A HDWE INTERRUPT DOES NOT EXIT
  THROUGH THE GATE, THOSE FUNCTIONS PROVIDED
  BY A GATE EXIT TO HANDLE PREEMPTS MUST BE
  PROVIDED HERE ALSO. !
CALL TEST_PREEMPT

! RESTORE LAST STATUS REGS !
POP  R8, @R15
POP  R7, @R15
LDM  @R5, R7, #2

! RESTORE NSP !
POP  R6, @R15
LDCTL NSP, R6
! RESTORE ALL REGSTERS !
LDM  R1, @R15, #16
ADD  R15, #32

! EXECUTE HARDWARE INTERRUPT RETURN !
IRET

ELSE ! NORMAL RETURN !
RET
FI
END GETWORK

```


! 00BE

273		
274		
275		
276		
277		
278		
279		
280		
281		
282		
283		
284		
285		
286		
287		
288		
289		
290		
291	00BE 6102	0002
292		
293		
294	00C2 6103	0006
295		
296		
297	00C6 0B03	FFFF
298	00CA 5F0F	00DA
299	00CE 7601	00CE
300	00D2 2100	0004
301	00D6 5F00	A900
302		
303		
304		
305	00DA 6134	00A2
306	00DE 6F04	0006
307		

!PAGE

```
ENTER_MSG_LIST PROCEDURE
!*****
! INSERTS POINTER TO MESSAGE
! FROM CURRENT_VP TO SIGNED_VP
! IN FIFO_MSG_LIST
!*****
! REGISTER USE:
! PARAMETERS:
!   R8(R9):MSG (INPUT)
!   R1: SIGNED_VP (INPUT)
!   LOCAL VARIABLES:
!   R2: CURRENT_VP
!   R3: FIRST_FREE_MSG
!   R4: NEXT_FREE_MSG
!   R5: NEXT_Q_MSG
!   R6: PRESENT_Q_MSG
!*****
ENTRY
LD R2, VPT.RUNNING_LIST

! GET FIRST MSG FROM FREE_LIST
LD R3, VPT.FREE_LIST

! * * * * * DEFUG * * * * *
CP R3, #NIL
IF EQ THEN
  LDA R1, $
  LD R0, #MSG_LIST_OVERFLOW
  CALL MONITOR
FI
! * * * * * END DEBUG * * * * *
```

```
LD R4, VPT.MSG.Q.NEXT_MSG(R3)
LD VPT.FREE_LIST, R4
```


308	00E2	763A	0090	! INSERT MESSAGE LIST INFORMATION !
309	00E6	2127	0010	LDA R10, VPT.MSG_Q.MSG(R3)
310	00EA	BAS1	07A0	LD R7, #SIZEOF MESSAGE
311	00EE	6F32	00A0	LDIRB QR10, QR8, R7
312				LD VPT.MSG_Q.SENDER(R3), R2
313				
314	00F2	6115	001E	! INSERT MSG IN MSG_LIST !
315				LD R5, VPT.VP.MSG_LIST(R1)
316				
317	00F6	0B05	FFFF	CP R5, #NIL
318	00FA	5F0E	0106	IF EQ ! MSG_LIST IS EMPTY ! THEN
319				! INSERT MSG AT TOP OF LIST !
320	00FE	6F13	001E	LD VPT.VP.MSG_LIST(R1), R3
321				
322	0102	5E08	011E	ELSE ! INSERT MSG IN LIST !
323				MSG_Q_SEARCH:
324				DO ! WHILE NOT END OF LIST !
325	0106	0B05	FFFF	CP R5, #NIL
326	010A	5E0E	0112	IF EQ ! END OF LIST ! THEN
327	010E	5E0E	011A	EXIT FROM MSG_Q_SEARCH
328				FI
329				
330				! GET NEXT LINK !
331	0112	A156		LD R6, R5
332	0114	6105	00A2	LD R5, VPT.MSG_Q.NEXT_MSG(R6)
333	0118	E8F6		OD
334				! INSERT MSG IN LIST !
335	011A	6F63	00A2	LD VPT.MSG_Q.NEXT_MSG(R6), R3
336				FI
337	011E	6F35	00A2	LD VPT.MSG_Q.NEXT_MSG(R3), R5
338	0122	9E08		RET
339	0124			END ENTER MSG_LIST
340				!PAGE

! 0124

```
341 GET_FIRST_MSG PROCEDURE
342 ! *****!
343 ! REMOVES MSG FROM MSG_LIST
344 ! AND PLACES ON FREE_LIST.
345 ! RETURNS SENDER'S MSG AND
346 ! VP_ID
347 ! *****!
348 ! REGISTER USE:
349 ! PARAMETERS:
350 ! R8(R9): MSG POINTER (INPUT)
351 ! R1: SENDER VP (RETURNED)
352 ! LOCAL VARIABLES
353 ! R2: CURRENT VP
354 ! R3: FIRST_MSG
355 ! R4: NEXT_MSG
356 ! R5: NEXT_FREE_MSG
357 ! R6: PRESENT_FREE_MSG
358 ! *****!
359 ENTRY
360 LD R2, VPT.RUNNING_LIST
361
362 ! REMOVE FIRST MSG FROM MSG_LIST !
363 LD R3, VPT.VP.MSG_LIST(R2)
364
365 ! * * * * * DEBUG * * * * !
366 CP R3, #NIL
367 IF EQ THEN
368 LD R2, #MSG_LIST_EMPTY
369 LDA R1, $
370 CALL MONITOR
371 FI
372 ! * * * * * END DEBUG * * * * !
373 LD R4, VPT.MSG_Q.NEXT_MSG(R3)
374 LD VPT.VP.MSG_LIST(R2), R4
375 ! PAGE
```


!			
0148	6105	0006	376
0140	0B05	FFFF	377
0150	5E0E	0162	378
0154	6F03	0006	379
0158	4D35	00A2	380
0150	FFFF		381
015E	5E08	017E	382
			383
			384
			385
			386
0162	0B05	FFFF	387
0166	5E0E	016E	388
016A	5E28	0176	389
			390
016E	A156		391
0170	6165	00A2	392
0174	E8F6		393
			394
			395
0176	6F63	00A2	396
017A	6F35	00A2	397
			398
			399
			400
017E	6131	00A0	401
0182	763A	0090	402
0186	2107	0010	403
018A	BAA1	0780	404
018F	9F08		405
0190			406
			407
			408

```

! INSERT MESSAGE IN FREE LIST !
LD R5, VPT.FREE_LIST
CP R5, #NIL
IF EQ ! FREE_LIST IS EMPTY ! THEN
! INSERT AT TOP OF LIST !
LD VPT.FREE_LIST, R3
LD VPT.MSG_Q.NEXT_MSG(R3), #NIL

ELSE ! INSERT IN LIST !
FREE_Q_SEARCH:
DO
CP R5, #NIL
IF EQ ! END OF LIST ! THEN
EXIT FROM FREE_Q_SEARCH
FI ! GET NEXT MSG !
LD R6, R5
LD R5, VPT.MSG_Q.NEXT_MSG(R6)
OD

! INSERT IN LIST !
LD VPT.MSG_Q.NEXT_MSG(R6), R3
LD VPT.MSG_Q.NEXT_MSG(R3), R5
FI ! GET MESSAGE INFORMATION:
.(RETURNS R1: SENDING_VP) !
LD R1, VPT.MSG_Q.SENDER(R3)
LDA R10, VPT.MSG_Q.MSG(R3)
LD R7, #SIZEOF MESSAGE
LDIRB R08, R10, R7
RET
END GET_FIRST_MSG

```


! 0190

```
409 MAKE_READY PROCEDURE
410 !*****
411 ! INSERTS SCHEDULE VP ID INTO !
412 ! READY LIST IAW PRIORITY AND !
413 ! PUTS IT IN READY STATE. !
414 !*****
415 ! REGISTER USE: !
416 ! PARAMETERS: !
417 ! R1: SIGNALFD VP (INPUT) !
418 ! LOCAL VARIABLES !
419 ! R2: SIG_VP.PRI !
420 ! R3: PRESENT_VP !
421 ! R4: NEXT_VP !
422 !*****
423 ENTRY
424 LD R1, VPT.RUNNING_LIST
425 ! * * * DEBUG * * * !
426 CP R4, #NIL
427 IF EQ ! LIST IS EMPTY ! THEN
428 LD R0, #READY_LIST_EMPTY
429 LDA R1, $
430 CALL MONITOR
431 FI
432 ! * * * END DEBUG * * * !
433
434 LD R2, VPT.VP.PRI (R1)
435
436 CP R2, VPT.VP.PRI(R4)
437 IF GT ! SIG_VP.PRI > READY_VP.PRI ! THEN
438 ! INSERT AT FRONT OF LIST !
439 LD VPT.VP.NEXT_READY_VP(R1), R4
440 LD VPT.READY_LIST, R1
441 !PAGE
```


!	01BC 5E08	01E8	442	ELSE ! INSERT IN LIST !
			443	
	01C0 0E04	FFFF	444	READY LIST SEARCH:
	01C4 5E0E	01CC	445	DO ! WHILE NOT END OF LIST !
	01C8 5E08	01E0	446	CP R4, #NIL
			447	IF EQ ! IF END OF LIST ! THEN
			448	EXIT FROM READY LIST SEARCH
			449	FI
	01CC 4B42	0012	450	
	01D0 5E02	01D8	451	CP R2, VPT.VP.PRI (R4)
	01D4 5E08	01E0	452	IF GT ! SIG_VP.PRI > PRESENT_VP.PRI ! THEN
			453	EXIT FROM READY LIST SEARCH
			454	FI
	01D8 A143		455	
	01DA 6134	001C	456	! GET NEXT LINK !
	01DE E8F0		457	LD R3,R4
	01E0 6F14	001C	458	LD R4, VPT.VP.NEXT_READY_VP(R3)
	01E4 6F31	001C	459	OD
			460	! INSERT SIG_VP IN LIST !
			461	VPT.VP.NEXT_READY_VP(R1), R4
			462	LD VPT.VP.NEXT_READY_VP(R3), R1
			463	
			464	FI
	01F8 4D15	0014	465	
	01EC 0001		466	! CHANGE STATE TO READY !
	01FF 9E0E		467	LD VPT.VP.STATE(R1), #READY
	01F0		468	
			469	RET
			470	END MAKE_READY
			471	!PAGE


```

472 ! * * INNER TRAFFIC CONTROL ENTRY POINTS * * !
473 GLOBAL
474 $SECTION ITC_GLB_PROC
475
476 HARDWARE_PREEMPT_LABEL
477
478 WAIT
479 ! ***** PROCEDURE ***** !
480 ! INTRA_KERNEL_SYNC/COM_PRIMITIVE !
481 ! INVOKED BY KERNEL PROCESSES !
482 ! ***** !
483 ! PARAMETERS !
484 ! RE(R9): MSG POINTER (INPUT) !
485 ! R1: SENDING_VP (RETURN) !
486 ! GLOBAL VARIABLES !
487 ! R14: DER (PARAM TO GETWORK) !
488 ! LOCAL VARIABLES !
489 ! R2: CURRENT_VP (RUNNING) !
490 ! R3: NEXT_READY_VP !
491 ! R4: LOCK_ADDRESS !
492 ! ***** !
493 ENTRY
494 ! LOCK_VPT !
495 LDA R4, VPT_LOCK
496 CALL SPIN_LOCK ! (R4:~VPT_LOCK) !
497 ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP !
498
499 LD R2, VPT_RUNNING_LIST
500 LD R3, VPT.VP.NEXT_READY_VP(R2)
501
502 CP VPT.VP.MSG_LIST(R2), #NIL
503
504 IF EQ ! CURRENT_VP'S MSG_LIST IS EMPTY ! THEN
505 ! REMOVE CURRENT_VP FROM READY_LIST !

```

! PAGE


```

!
001A 0E03 FFFF
001E 5E0E 002E
0022 2100 0003
0026 7601 0026
002A 5F00 A900

002E 6F03 0004
0032 4D25 001C
0036 FFFF

0038 4D25 0014
003C 0002

003E 612E 0010
0042 93F8
0044 5F00 0000
004E 97FE

004A 5F00 0124
004E 4D08 0000

0052 9F06
0054

506 ! ** * * DEBUG * * * * !
507 CP R3, #NIL
508 IF EQ THEN
509 LD R0, #READY_LIST_EMPTY
510 LDA R1, $
511 CALL MONITOR
512 FI
513 ! ** * * END DEBUG * * * * !
514
515 VPT.READY_LIST, R3
516 VPT.VP.NEXT_READY_VP(R2), #NIL

517 ! PUT IT IN WAITING STATE !
518 LD VPT.VP.STATE(R2), #WAITING

520 ! SFT DBR !
521 LD R14, VPT.VP.DBR(R2)
522 ! SCHEDULE FIRST ELIGIBLE READY VP !
523 PUSH @R15,R8 !SAVE MSG POINTER!
524 CALL GETWORK I(R14: DBR) !
525 PCP R8,@R15
526 FI
527 ! GET FIRST MSG ON CURRENT VP'S MSG LIST !
528 CALL GET_FIRST_MSG !COPIES MSG IN MSG ARRAY!
529 !RETURNS R1:SENDER_VP !
530
531 ! UNLOCK VPT !
532 CLR VPT.LOCK
533
534 ! RETURN: R1:SENDER_VP !
535 RET
536 END WAIT
537
538 !PAGE

```



```

540 SIGNAL PROCEDURE
541 !*****!
542 ! INTRA_KERNEL_SYNC /COM PRIMITIVE !
543 ! INVOKED BY KERNEL PROCESSES !
544 !*****!
545 ! REGISTER USE:
546 ! PARAMETERS:
547 ! RE(R9): MSG POINTER (INPUT)
548 ! R1: SIGNED VP_ID (INPUT)
549 ! GLOBAL VARIABLES
550 ! R14: DBR (PARAM TO GETWORK)
551 ! LOCAL VARIABLES:
552 ! R1: SIGNED VP
553 ! R2: CURRENT VP
554 ! R4: VPT.LOCK ADDRESS
555 !*****!
556 ENTRY
557 ! LOCK VPT !
558 LDA R4, VPT.LOCK
559 CALL SPIN_LOCK ! (R4:~VPT.LOCK) !
560 ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !
561 ! PLACE MSG IN SIGNED VP'S MSG LIST !
562 CALL ENTER_MSG_LIST ! (R2:MSG PTR,
563 R1:SIGNED VP) !
564
565 CP VPT.VP.STATE(R1), #WAITING
566
567 IF EQ ! SIGNED_VP IS WAITING ! THEN
568
569 ! WAKE IT UP AND MAKE IT READY !
570 CALL MAKE_READY ! (R1: SIGNED_VP) !
571
572 ! PUT CURRENT_VP IN READY_STATE !
573 LD R2, VPT.RUNNING_LIST
574 LD VPT.VP.STATE(R2), #READY
575
576 ! PAGE

```


!		575	! SET DPR !
007E 612E 0010		576	LD R14, VPT.VP.DRR(R2)
		577	
007C 5F00 0000		578	! SCHEDULE FIRST ELGIBLE READY VP !
		579	CALL GETWORK !(R14: DRR) !
		580	FI
		581	
0080 4D08 0000		582	! UNLOCK VPT !
		583	CLR VPT.ILOCK
0084 9E08		584	
008C		585	RET
		586	END SIGNAL
		587	!PAGE


```

588 SET_PREEPT PROCEDURE
589 !*****!
590 ! SETS PREEPT INTERRUPT ON!
591 ! TARGET_VP. CALLED BY TC_!
592 ! ADVANCE.
593 !*****!
594 ! REGISTER USE:
595 ! PARAMETERS:
596 ! R1:TARGET_VP_ID (INPUT)
597 ! LOCAL_VARIABLES
598 ! R1: VP_INDEX
599 !*****!
600 ENTRY
601 ! NOTE: DESIGNED AS SAFE SEQUENCE SO VPT NEED
602 ! NOT BE LOCKED. !
603
604 ! CONVERT VP_ID TO VP_INDEX !
605 LDK R0, #0
606 MULT R0, #SIZEOF_VP_TABLE
607 ! THIS LEAVES VP_INDEX IN R1 !
608
609 ! TURN ON TGT_VP_PREEPT FLAG !
610 LD VPT_VP_PREEPT(R1), #ON
611
612 ! ** IF TARGET_VP NOT LOCAL
613 ( NOT BOUND TO THIS_CPU )
614 [IE, IF <<CPU_SEG>>CPU_ID<>VPT_VP_PHYS_CPU(R1)]
615 THEN SEND_HARDWARE_PREEPT_INTERRUPT_TO
616 VPT_VP_CPU(R1). ** !
617
618 RET
619 END SET_PREEPT
620 !PAGE

```


621			
622			
623			
624			
625			
626			
627			
628			
629			
630			
631			
632			
633			
634			
635			
636			
637			
638	0094	7604	0000'
639	009E	5F00	0154'
640			
641			
642			
643	009C	6102	0002'
644			
645			
646			
647			
648			
649	00A4	2103	0060'
650	00A8	6135	0012'
651	00AC	6F25	0010'
652			
653			
654	00F0	4D25	0016'
	00R4	FFFF	

655 !PAGE

```

IDLE          PROCEDURE
!*****!
! LOADS IDLE DBR ON
! CURRENT VP. CALLED BY
! TC GETWORK.
!*****!
! REGISTER USE
! GLOBAL VARIABLE
! R14: DBR
! LOCAL VARIABLES:
! R2: CURRENT VP
! R3: TEMP VAR
! R4: VPT.LOCK ADDR
! R5: TEMP
!*****!
ENTRY
! LOCK VPT !
LDA R4, VPT.LOCK
CALL SPIN_LOCK ! (R4: VPT.LOCK) !
! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP!

! GET CURRENT VP !
LD R2, VPT.RUNNING_LIST

! SET DBR !
LD R14, VPT.VP.DBR(R2)

! LOAD IDLE DBR ON CURRENT VP !
LD R3, #IDLE_VP*SIZECF VP_TABLE
LD R5, VPT.VP.DBR(R3)
LD VPT.VP.DBR(R2), R5

! TURN ON CURRENT VP'S IDLE FLAG !
LD VPT.VP.IDLE_FLAG(R2), #ON

```


656			! SET VP TO READY STATE !
657	00R6 4D25 0014	00BA 0001	LD VPT.VP.STATE(R2), #READY
658			
659			! SCHEDULE FIRST ELIGIBLE READY VP !
660	00RC 5F00 0000		CALL GETWORK !(R14: DBR) !
661			
662			! UNLOCK VPT !
663	00C0 4D08 0000		CLR VPT.LOCK
664			
665	00C4 9E08		RFT
666	00C6		END IDLE
667			!PAGE


```

SWAP_VDBR PROCEDURE
!*****!
! LOADS NFW DRR ON !
! CURRENT VP. CALLED BY !
! TC_GETWORK. !
!*****!
! REGISTER USE !
! PARAMETERS !
! R1: NEW_DRR (INPUT) !
! GLOBAL VARIABLES !
! R14: DRR !
! LOCAL VARIABLES !
! R2: CURRENT VP !
! R4: VPT.LOCK ADDR !
!*****!
ENTRY
! LOCK VPT !
LDA R4, VPT.LOCK
CALL SPIN_LOCK ! (R4:~VPT.LOCK) !
! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !
! GET CURRENT VP !
LD R2, VPT.RUNNING_LIST
! * * * DEFUG * * * !
CP VPT.VP.MSG_LIST(R2), #NIL
IF NE ! MSG WAITING ! THEN
LD R0, #SWAP_NOT_ALLOWED
LDA R1, $ !PC!
CALL MONITOR
FI
! * * * END DEBUG * * * !
! SET DRR !
LD R14, VPT.VP.DRR(R2)

```


!					
00EC 6F21	0010'	702	! LOAD NEW DBR ON CURRENT VP !		
		703	LD VPT.VP.DBR(R2), R1		
		704			
00F0 4D25	0016'	705	! TURN OFF IDLE FLAG !		
00F4 0000		706	LD VPT.VP.IDLE_FLAG(R2), #OFF		
		707			
00F6 4D25	0014'	708	! SET VP TO READY STATE !		
00FA 0001		709	LD VPT.VP.STATE(R2), #READY		
		710			
00FC 5F00	0000'	711	! SCHEDULE FIRST ELGIFLE READY VP !		
		712	CALL GETWORK ! (R14:DBR) !		
		713			
0100 4D08	0000'	714	! UNLOCK VPT !		
		715	CLR VPT.LOCK		
0104 9E08		716			
0106		717	RET		
		718	END SWAP_VDBR		
		719			!PAGE


```

720 TEST__PREEMPT
721
722 PROCEDURE
723 *****!
724 ! TESTS FOR PREEMPT INTERRUPT !
725 ! FLAG AND HANDLES INTERRUPT !
726 ! IF FLAG IS SET. !
727 ! INVOKED UPON EVERY EXIT FROM !
728 ! KERNEL. !
729 *****!
730 ! REGISTER USE !
731 ! LOCAL VARIABLES !
732 ! R1: PREEMPT_INT_FLAG !
733 ! R2: CURRENT_VP !
734 *****!
735 ENTRY
736
737 TEST_FLAG:
738 DO ! WHILE CURRENT_VP'S PREEMPT FLAG IS ON !
739
740 !NOTE: NEXT TWO STATEMENTS MAY NOT BE RACE FREE.
741 LOCK MAY BE REQUIRED FOR MULTIPROCESSOR SYS!
742
743 ! GET CURRENT_VP !
744 LD R2, VPT.RUNNING_LIST
745
746 ! TEST PREEMPT INTERRUPT FLAG !
747 LD R1, VPT.VP.PREEMPT(R2)
748 CP R1, #OFF
749 IF EQ ! PREEMPT FLAG IS OFF ! THEN
750 EXIT FROM TEST_FLAG
751 FI
752
753 ! *** VIRTUAL PREEMPT HANDLER *** !
754 ! *** NOTE: SAFE SEQUENCE AND DOES NOT REQUIRE
755 VPT TO BE LOCKED. *** !

```

```

0106 6102 0002'
010A 6121 0018'
010F 0F01 0000'
0112 5E0E 011A'
0116 5E08 0126'

```


011A 4D25	0018'	754	! RESET PREEMPT FLAG !
011E 0000		755	LD VPT.VP.PREEMPT(R2), #OFF
0120 5F00	0000*	756	
		757	! SIMULATE PREEMPT INTERRUPT !
		758	CALL TC_PREEMPT_HANDLER
		759	! NOTE: THIS JUMP TO TRAFFIC CONTROL
		760	IS USED ONLY IN THE CASE OF A PREEMPT INTERRUPT
		761	AND SIMULATES A HARDWARE INTERRUPT. ** !
		762	
0124 E8F0		763	! *** END VIRTUAL PREEMPT HANDLER *** !
		764	OD
0126 9E08		765	
0128		766	! RETURN TO GATEKEEPER !
		767	RET
		768	
		769	END TEST..PREEMPT
		770	!PAGE

! 0126

771		RUNNING_VP	PROCEDURE
772		!	*****!
773		!	! CALLED PY TRAFFIC CONTROL.
774		!	! RETURNS VP_ID. RESULT IS VALID!
775		!	! ONLY WHILE_APT IS LOCKED.
776		!	! *****!
777		!	! REGISTER USE
778		!	! PARAMETERS
779		!	! R1: VP_IL (RETURNED)
780		!	! LOCAL VARIABLES
781		!	! RR0: DIVIDEND
782		!	! R0: REMAINDER
783		!	! R1: QUOTIENT
784		!	! *****!
785		ENTRY	
786		!	! LOCK VPT !
787	0128 7604	LDA	R4, VPT.LOCK
788	012C 5F00	CALL	SPIN_LOCK ! (R4: VPT.LOCK) !
789		!	! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP !
790	0130 6101	LD	R1, VPT.RUNNING_LIST
791	0134 FD00	LDK	R0, #0
792		!	! CONVERT VP_INDEX TO VP_ID !
793	0136 1P00	DIV	RR0, #SIZEOF_VPT_TABLE
794		!	! * * * DFPUG * * * !
795	013A 0B00	CP	R0, #0
796	013E 5F06	IF	NE, !REMAINDER <> 0 ! THEN
797	0142 2100	LD	R0, #VP_INDEX_ERROR
798	0146 7601	LDA	R1, \$
799	014A 5F00	CALL	MONITOR
800		FI	
801		!	! * * * END DEBUG * * * !
802	014E 4D08	CLR	VPT.LOCK
803	0152 9E08	RET	
804	0154	END	RUNNING_VP
805			!PAGE

! 0154

806		
807		
808		
809		
810		
811		
812		
813		
814		
815		
816		
817		
818		
819		
820		
821	0000	
822	0168	
823	0000	
824	0160	
825	A900	
826		
827		
828		
829		
830	0040	
831	016A	
832		
833		
834		
835		
836		
837		
838		
839		

!PAGE

```
SPIN_LOCK PROCEDURE
!*****!
! USES SPIN_LOCK MECH. !
! LOCKS UNLOCKED DATA !
! STRUCTURE (POINTED TO !
! BY INPUT PARAMETER). !
!*****!
!REGISTER USE !
! PARAMETERS !
! R4: LOCK ADDR (INPUT)!
!*****!
ENTRY
! NOTE: SINCE ONLY ONE PROCESSOR CURRENTLY IN
      SYSTEM, LOCK NOT NECESSARY. ** !
      ! * * * DEBUG * * * !
      CP @R4, #OFF
      IF NE ! NOT UNLOCKED ! THEN
      LD R0, #UNAUTH_LOCK
      LDA R1, $
      CALL MONITOR
      FI
      ! * * * END DEBUG * * * !

TEST_LOCK:
! DO WHILE STRUCTURE LOCKED !
TSET @R4
JR MI, TEST_LOCK
! ** NOTE - SEE PLZ/ASM MANUAL
FOR RESTRICTIONS ON
USE OF TSET. ** !

RET

END SPIN_LOCK
```


! 016E

840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860

016E 7E04 0000'
0172 5F00 0154'
0176 6102 0002'
017A 6121 001A'
017E 4D08 0000'
0182 9E08
0184

```
ITC_GET_CPU_NO  
!*****PROCEDURE*****!  
! FIND CURRENT CPU_NO !  
! CALLED BY DIST MMGR !  
!*****!  
! REGISTER USE !  
! R1: CPU_NO (RETURNED) !  
! R2: VP_INDEX (LOCAL) !  
!*****!  
  
ENTRY  
    LDA    R4,VPT.LOCK  
    CALL   SPIN_LOCK      ! R4: ~VPT.LOCK !  
    LD     R2,VPT.RUNNING_LIST  
    LD     R1,VPT.VP.PHYS_PROCESSOR(R2)  
    CLR    VPT.LOCK  
    RET  
END ITC_GET_CPU_NO
```

! PAGE

! 0184

```

261 ITC_GET_SEG_PTR          PROCEDURE
262 !*****!
263 ! JFTS BASE ADDRESS OF SEGMENT !
264 ! INDICATED. !
265 !*****!
266 ! REGISTER USE: !
267 ! R0:SEG BASE ADDRESS(RET) !
268 ! R1:SEG NR (INPUT) !
269 ! R2:RUNNING_VP (LOCAL) !
270 ! R3:DBR_VALUE (LOCAL) !
271 ! R4:VPT_LOCK !
272 !*****!
273
274 ENTRY
275 LDA R4,VPT_LOCK
276 CALL SPIN_LOCK !R4:VPT_LOCK!
277 LD R2,VPT.RUNNING_LIST
278 LD R3,VPT.VP.DBR(R2)
279 CLR VPT_LOCK
280 MULT R0,#4
281 LD R0,R3(R1) !NOTE: DBR (NOT DER_NO)
282 RET USED HERE!
283
284 END ITC_GET_SEG_PTR
285 !PAGE
```

```

0184 7604 0000
0188 5F00 0154
018C 6102 0002
0190 6123 0010
0194 4D08 0000
0198 1900 0004
019C 7130 0100
01A0 9E08
01A2
```


!	01A2	886	K_LOCK	PROCEDURE
		887	*****!	*****!
		888	! STUB FOR WAIT LOCK!	! STUB FOR WAIT LOCK!
		889	*****!	*****!
		890	! R4: LOCK (INPUT) !	! R4: LOCK (INPUT) !
		891	*****!	*****!
		892		
		893	ENTRY	
		894	CALL SPIN_LOCK	
		895	RET	
		896	END K_LOCK	
		897		
		898		
		899		
		900	K_UNLCK	PROCEDURE
		901	*****!	*****!
		902	! STUB FOR WAIT UNLOCK !	! STUB FOR WAIT UNLOCK !
		903	*****!	*****!
		904	! R4: LOCK (INPUT) !	! R4: LOCK (INPUT) !
		905	*****!	*****!
		906	ENTRY	
		907	CLR @R4	
		908	RET	
		909	END K_UNLOCK	
		910		
		911	END INNER_TRAFFIC_CONTROL	
		912		!PAGE

01A2 5F00 0154
01A6 9E0E
01A8

01A8 0D4E
01AA 9E0E
01AC

2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35

TC MODULE

! 11 SEP 1980 !

CONSTANT

! ***** DEBUG CODES ***** !
BLOCKED_LIST_ERROR := 0
READY_LIST_ERROR := 1
RUNNING_LIST_ERROR := 2

! ***** SYSTEM PARAMETERS ***** !
NR_PROCESSES := 4
NR_MMU_REG := 64
NR_VP := 4
NR_AVAIL_VP := 2
STACK_SEG := 1
STACK_SEG_SIZE := %100

! * * OFFSETS (FROM TOP OF STACK) * * !
PROCESS_ID := STACK_SEG_SIZE-%10

! ***** SYSTEM CONSTANTS ***** !
TRUE := 1
FALSE := 0
ON := %FFFF
OFF := 0
RUNNING := 0
READY := 1
BLOCKED := 2
IDLE1 := %DDDD
NIL := %FFFF
INVALID := %EEEE
MCNITOR := %A9C2

! HPUG ENTRY !

!PAGE

36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57

```
TYPE
  AP_POINTER WORD
  ADDRESS WORD
  EVENT_TABLE RECORD
  [ HANDLE WORD
    EVENT WORD
    TICKET WORD
    FILLER_2 ARRAY [5 WORD]
  ]
  AP_TABLE RECORD
  [ DBR_NO SHORT_INTEGER
    FILLER_0 BYTE
    SAC LONG
    PRI INTEGER
    STATE INTEGER
    NEXT_AP AP_POINTER
    FILLER_1 ARRAY [2 WORD]
    EVENTCOUNT EVENT_TABLE
  ]
  RUNNING_ARRAY ARRAY [NR_AVAIL_VP WORD]
```

IPAGE


```

58 EXTERNAL
59   SPIN_LOCK
60   SFT_PREENPT
61   SWAP_VDBR
62   IDLE
63   RUNNING_VP
64   MM_GET_DBR_VALUE
65   K_LOCK
66   K_UNLOCK
67
68   $SECTION TC_DATA
69   INTERNAL
70   APT_RECORD
71   [ $SUCCESS_CODE
72     LOCK
73     RUNNING_LIST
74     READY_LIST
75     BLOCKED_LIST
76     FILLER
77     AP
78   ]
79   !PAGE

```

```

PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE

```

```

WORD
WORD
RUNNING_ARRAY
WORD
WORD
ARRAY[2 WORD]
ARRAY [NR_PROCESSES AP_TABLE]

```

0000

GLOBAL
\$SECTION TC_GLB_PROC

TC_GETWORK

0000

PROCEDURE

```
!*****!  
! LOADS NEXT READY DBR !  
! ON CURRENT VP.      !  
!*****!  
! REGISTER USE        !  
! LOCAL VARS          !  
! R1: CURRENT VP_ID   !  
! R2: READY_AP        !  
! R3: VP_PTR          !  
!*****!
```

ENTRY

```
! FIND FIRST READY PROCESSOR !  
LD R2, APT.READY_LIST  
READY_AP_SEARCH:  
DC ! WHILE NOT (ENT OF LIST OR READY_PROCESS) !
```

```
CP R2, #NIL  
IF EQ ! IF NO READY PROCESSES ! THEN  
EXIT FROM READY_AP_SEARCH  
FI
```

```
CP APT.AP.STATF(R2), #READY
```

```
IF EQ ! IF PROCESS READY ! THEN  
EXIT FROM READY_AP_SEARCH  
FI
```

```
! GET NEXT READY AP !  
LD R3, APT.AP.NEXT_AP(R2)  
LD R2, R3  
OD
```

!PAGE

80			
81			
82			
83			
84			
85			
86			
87			
88			
89			
90			
91			
92			
93			
94			
95			
96			
97			
98			
99			
100			
101			
102			
103			
104			
105			
106			
107			
108			
109			
110			
111			
112			
113			
114			

!	0026	0102	FFFF	115	CP	R2,#NIL
	002A	5E0E	003C	116	IF EQ !	IF NO PROCESSES READY ! THEN
				117	!	LOAD IDLE PROCESS !
	002E	4D15	0004	118	LD	APT.RUNNING_LIST(R1), #IDLE1
	0032	DDDD		119	CALL	IDLE
	0034	5F00	0000*	120	ELSE	
	003E	5F08	0054	121	!	LOAD FIRST READY AP !
	003C	6F12	0004	122	LD	APT.RUNNING_LIST(R1), R2
	0040	4D25	0018	123	LD	APT.AP.STATE(R2), #RUNNING
	0044	0000				
	0046	6029	0010	124	LDB	RL1, APT.AP.DBR_NO(R2)
	004A	5F00	0000*	125	CALL	MM_GET_DBR_VALUE ! (R1:DBR_NO)!
				126		! (RETURNS:R2:IPR)!
	004E	A121		127	LD	R1,R2 !DBR VALUE (ADDRESS)!
	0050	5F00	0000*	128	CALL	SWAP_VDBR ! (R1:DBR)!
				129	FI	
	0054	9E08		130	RET	
	0056			131	END	TC_GETWORK
				132		!PAGE

! 0056

133			
134			
135			
136			
137			
138			
139			
140			
141			
142			
143			
144			
145			
146			
147			
148	0056	5F00	0000*
149			
150			
151	005A	6112	0004'
152			
153			
154	005E	0F02	DDDD
155	0062	5E26	006C'
156	0066	4D25	0018'
	006A	0001	
157			
158			
159			
160	006C	5F00	0000'
161			
62			
163			
164	0070	9E08	
165	0072		
166			
167			
168			
169			
170			

```

TC_PREEMPT_HANDLER PROCEDURE
!*****!
! LOADS FIRST READY AP
! IN RESPONSE TO PREEMPT
! INTERRUPT
!*****!
! GLOBAL (TC) VARIABLES
! R12: CURRENT PROCESS
! R13: SEG_BASE_ADDR
! R14: DEF_NO
!*****!
ENTRY
! ** CALL WAIT_LOCK (APT^.LOCK) **!
! ** RETURNS WHEN PROCESS HAS LOCKED APT **!
! GET RUNNING VP ID !
CALL RUNNING_VP ! (RETURNS: R1:VP_ID)!

! GET AP !
LD R2, APT.RUNNING_LIST(R1)

! IF NOT AN IDLE PROCESS, SET IT TO READY !
CP R2, #IDLE1
IF NE ! NOT IDLE ! THEN
LD APT.AP.STATE(R2), #READY
FI

! LOAD FIRST READY PROCESS !
CALL TC_GETWORK
! ** CALL WAIT_UNLOCK (APT^.LOCK) **!
! ** RETURNS WHEN PROCESS HAS UNLOCKED APT **!
! ** AND ADVANCED ON THIS EVENT **!
RET
END TC_PREEMPT_HANDLER

```

! PAGE

! 008C

190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214

008C 76E4 0002'
0090 5F00 0000*
0094 5F00 0000*
009E 6115 0004'
009C 6059 0010'
00A0 4D08 0002'
00A4 9EE8
00A6

```
TC_GET_DBR_NO      PROCEDURE
!*****!
! SEARCHES APT FOR CURRENT VALUE!
! OF DBR_NO.
!*****!
! REGISTER USE:
! R1:CUR_VP_ID (LOCAL)
! R1:DBR_NO (RETURNED)
! R4:APT_LOCK (LOCAL)
! R5:PROCESS_ID (LOCAL)
!*****!

ENTRY
  LDA      R4,APT_LOCK
  CALL     K_LOCK !R4:~APT_LOCK!
  CALL     RUNNING_VP ! (RETURNS:R1:VP_ID) !
  LD       R5,APT_RUNNING_LIST(R1)
  LDB      R11,APT.AP.DBR_NC(R5)
  CLR      APT_LOCK
  RET
END TC_GET_DBR_NO

END TC
```



```
2 IDLE_PROCESS MODULE
3   ! VERS. 1.8 !
4   CONSTANT
5   WRITELN      := %0FC0
6   RETURN_TO_HPUG := %A902
7
8   EXTERNAL
9   SIGNAL
10  WAIT
11  IDLE
12  SET_PREENPT
13  SWAP_VDBR
14  TEST_PREENPT
15
16  INTERNAL
17
18  $SECTION IDLE_DATA
19  MSG ARRAY [* EYTE] := 'I ** IDLE PROCESS IS RUNNING! ** %R'
```

```
0000 49 20 2A
0003 2A 20 40
0006 44 40 50
0009 20 43 53
000C 4F 53 55
000F 53 55 4E
0012 49 53 4E
0015 52 49 21
0018 4E 21 2A
001B 47 2A 20
001E 2A 2A 20
0021 0D
```


!

0000	21	SECTION IDLE PROC
	22	IDLE_MAIN PROCEDURE
	23	
0000	24	ENTRY
0000	25	LD R2, #MSG
0004	26	CALL WRITEIN
0008	27	CALL RETURN_TO_HRUG
000C	28	RET
	29	
	30	TEST ENTRY:
	31	! PARAMETER R10: CALLS PROCEDURES !
	32	IF R10
	33	CASE #0 THEN CALL IDLE
	34	CASE #1 THEN CALL SWAP_VDER
	35	CASE #2 THEN CALL SET_PREEMPT
	36	CASE #3 THEN CALL TEST_PREEMPT
	37	ELSE
	38	CALL RETURN_TO_HRUG
	39	
	40	FI
	41	
	42	RET
	43	END IDLE_MAIN
	44	
	45	END IDLE_PROCESS
	46	!PAGE


```

2 MM_PROCESS      MODULE
3
4 ! VERS. 1.8      !
5
6 CONSTANT
7 WRITE           := %0FC8
8 WRITELN         := %0FC0
9 CR LF           := %0FD4
10 RETURN_TO_MONITOR := %A902
11
12 COUNT := 10
13 TIME := 500
14
15 SPACE := %20
16 DASH := %2D
17
18 IO_MGR          := %20
19 FILE_MGR        := %40
20 MEM_MGR         := %60
21 CREATE_ENTRY_CODE := 50
22 INVALID_MMGR_CODE := 60
23 DELFT_ENTRY_CODE := 51
24 ACTIVATE_SEG_CODE := 52
25 DEACTIVATE_SEG_CODE := 53
26 SWAP_IN_SEG_CODE := 54
27 SWAP_OUT_SEG_CODE := 55
28 SUCCEEDED
29
30 ! PAGE

```

```

! TYWR MON CALL !
! PUTMSG MON CALL !
! MCN CALL      !
! HBUG REENTRY  !

```



```

31 TYPE
32 ADDRESS
33 SEG_DESC_REG RECORD [BASE_ADDR ADDRESS
34 LIMIT BYTE
35 ATTRIBUTE BYTE]
36 MMU RECORD [SDR_ARRAY [64 SEG_DESC_REG]]
37 !BLKS_USED WORD
38 MAX_ELKS WORD]]
39 !NOTE: LAST TWO MMU COMPONENTS LEFT
40 OFF FOR CONVENIENCE SINCE ARE NOT
41 USED FOR THE SEG MGR DEMO!
42 EXTERNAL
43 SIGNAL
44 WAIT
45
46 GLOBAL
47 $SECTION MMU_DATA
48 MMU_IMAGE
49 $SECTION MM_DATA
50 G_AST_LOCK
51
52 !PAGE

```

0000

0000 0000

ARRAY [4 MMU]

WORD := 0

PROCEDURE
PROCEDURE

\$SECTION MMU_DATA
MMU_IMAGE
\$SECTION MM_DATA
G_AST_LOCK

!

```
53 INTERNAL
54
55 ! * * * MESSAGES * * * !
56 IO      ARRAY [* BYTE] := '%08(FOR IO)'

57 FM      ARRAY [* BYTE] := '%08(FOR FM)'

58 MM_MSG_1 ARRAY [* BYTE] := '%12KERNEL = SIGNALLER'

59 CREATE_MSG ARRAY [* BYTE] := '%10MM: CREATE_ENTRY'

60 DELETE_MSG ARRAY [* BYTE] := '%10MM: DELETE_ENTRY'

61 !PAGE
```

```
0002 08 28 46
0005 4F 52 20
0008 49 29 4F
000B 08 28 46
000E 4F 52 20
0011 46 29 4D
0014 12 4B 45
0017 52 4E 45
001A 4C 20 3D
001D 20 53 49
0020 47 4E 41
0023 4C 4C 45
0026 52 4D 4D
0027 10 4D 43
002A 3A 20 41
002D 52 45 41
0030 54 45 5F
0033 45 4E 54
0036 52 59 4D
0038 10 4D 44
003E 3A 20 44
003F 45 4C 45
0041 54 45 5F
0044 45 4E 54
0047 52 59 4F
```


72	INTERNAL		
73			
74	\$SECTION MM_PROC		
75			
76	MM_MAIN PROCEDURE		
77	ENTRY		
78	DO !** DO FOREVER **!		
79	LD RE,MM_MSG_ARRAY[0]	0000F	009F
80	CALL WAIT	0004 5F00	0000*
81	SENDER, R1 !SAVE SIGNALING PROC #!	0008 6F01	00AC
82	R3,#50	000C 2103	0032
83	MM_PRINT BLANKS	0010 5F00	0178
84	R2,MM_MSG_1	0014 2102	0014
85	WRITELN	0018 5F00	0FC0
86	R1,SENDER	001C 6101	00AC
87	R1		
88	CASE #IO_MGR THEN LD R2,#IO	0020 0B01	0020
		0024 5E0E	0034
		0028 2102	0002
89	CALL WRITELN	002C 5F00	0FC0
90	CASE #FILE_MGR THEN LD R2,#FM	0030 5E08	0044
		0034 0B01	0040
		0038 5E0E	0044
		003C 2102	000B
91	CALL WRITELN	0040 5F00	0FC0
92			
93	MM_DELAY	0044 5F00	0144
94	CRIF	0048 5F00	0FD4
95	R3,#50	004C 2103	0032
96	MM_PRINT BLANKS	0050 5F00	0178
97	RH1,MM_MSG_ARRAY[0]	0054 6001	009B
98	R11,MM_MSG_ARRAY[1]	0058 6009	009C
99	!PAGE		

00C0	5F00	0FC0	117
00C4	5F00	0144	118
00C8	5F00	0FD4	119
00CC	2103	004F	120
00D0	5F00	0160	121
00D4	5F00	0FD4	122
00D8	6101	00AC	123
00DC	7608	009B	124
00E0	5F00	0000*	125
00E4	E8D		126
00E6	9E08		127
00E8			128
			129
			130
			131
00E8			132
			133
			134
00FE	7608	009B	135
00EC	0C85	0202	136
00F0	2102	0027	137
00F4	9E08		138
00F6			139
			140
			141
			142
00F6			143
			144
00F6	7608	009B	145
00FA	0C85	0202	146
00FE	2102	0038	147
2102	9E08		148
0104			149
			150
			151

!PAGE

```

CALL WRITELN
CALL MM_DELAY
CALL CRLF
LD R3,#75
CALL MM_PRINT_LINE
CALL CRLF
! ** SIGNAL (SENDER, 'DONE') ** !
LD R1, SENDER
LDA R8,MM_MSG_ARRAY[0]
CALL SIGNAL
OD ! ** REPEAT FOREVER ** !
RET
END MM_MAIN

```

PROCEDURE

CREATE_ENTRY

```

ENTRY
LDA R8,MM_MSG_ARRAY[0]
LDB QR8,#SUCCEEDED
LD R2,#CREATE_MSG
RET
END CREATE_ENTRY

```

PROCEDURE

DELETE_ENTRY

```

ENTRY
LDA R8,MM_MSG_ARRAY[0]
LDB QR8,#SUCCEEDED
LD R2,#DELETE_MSG
RET
END DELETE_ENTRY

```


!	0104		ACTIVATE	PROCEDURE
152			ENTRY	
153			LDA R8,MM MSG_ARRAY[0]	
154			LDA R9,RET_VALUES[0]	
155	0104	7608 009B'	LD R2,#16	
156	0108	7609 008B'	LDIRB QR8,QR9,R2	
157	010C	2102 0010	LD R2,#ACTIVATE_MSG	
158	0110	BA91 0280'	RET	
159	0114	2102 0049'	END ACTIVATE	
160	0118	9E08		
161	011A			
162			DEACTIVATE	PROCEDURE
163	011A		ENTRY	
164			LDA R8,MM MSG_ARRAY[0]	
165			LD R8,QR8,#SUCCEEDED	
166			LD R2,#DEACTIVATE_MSG	
167	011A	7608 009B'	RET	
168	011E	0C85 0202'	END DEACTIVATE	
169	0122	2102 0056'		
170	0126	9E08		
171	0128			
172			SWAP_IN	PROCEDURE
173	0128		ENTRY	
174			LDA R8,MM MSG_ARRAY[0]	
175			LDF QR8,#SUCCEEDED	
176	0128	7608 009B'	LD R2,#SWAP_IN_MSG	
177	012C	0C85 0202'	RET	
178	0130	2102 0065'	END SWAP_IN	
179	0134	9E08		
180	0136			
181				
182				
183				
184				! PAGE

!	0136		SWAP_OUT	PROCEDURE
0136	7608	009B	ENTRY	
013A	0CE5	0202	LDA	R2,MM_MSG_ARRAY[0]
013E	2102	0071	LDB	QR8,#SUCCEDED
0142	9E08		LD	R2,#SWAP_OUT_MSG
0144			RET	
			END SWAP_OUT	
0144			MM_DELAY	PROCEDURE
				!*****!
				! PRODUCES 2 SEC DELAY !
				!*****!
			ENTRY	
0144	2102	000A	LD	R2, #COUNT
0148	2101	01F4	LD	R1, #TIME
			DC	
014C	0B02	0000	CP	R2, #0
0150	5E0E	0158	IF EQ THEN	EXIT FI
0154	5E0E	015F		
0158	AB20		DEC	R2
015A	7B1D		MREQ	R1
015C	EF7		OD	
015E	9E08		RET	
0160			END MM_DELAY	
				! PAGE

214		MM_PRINT_LINE	PROCEDURE	! ***** !
215	0160			! PRINTS LINE LENGTH !
216				! SPEC IN R3. !
217				! ***** !
218				
219				
220		ENTRY		
221	0160	LDB	RL0, #DASH	
222		DO		
223	0162	CP	R3, #0	
224	0166	IF EQ THEN EXIT FI		
225	016A			
226	016E	CALL WRITE		
227	0172	DEC	R3	
228	0174	OD		
229	0176	RET		
230	017E	END MM_PRINT_LINE		
231	0178	MM_PRINT_BLANKS	PROCEDURE	! ***** !
232				! PRINTS NUMBER OF !
233				! BLANKS SPEC IN R3. !
234				! ***** !
235				
236		ENTRY		
237	0178	LDR	RL0, #SPACE	
238		DO		
239	017A	CP	R3, #0	
240	017E	IF EQ THEN EXIT FI		
241	0182			
242	0186	CALL WRITE		
243	018E	DEC	R3	
244	0190	OD		
245		RET		
246		END MM_PRINT_BLANKS		
247		END MM_PROCESS		
248		!PAGE		


```

2 FM_PROCESS      MODULE
3
4 ! VERS 1.8      !
5 CONSTANT
6 WRITE           := %0FC8      ! TYWR MON CALL !
7 WRITELN        := %0FC0      ! PUTMSG MON CALL !
8 CRLF           := %0FD4      ! MON CALL      !
9 RETURN_TO_MONITOR := %A902    ! HRUG PEENTRY  !
10
11 COUNT := 10
12 TIME  := 500
13
14 SPACE := %20
15 DASH  := %2D
16
17 IO_MGR      := %20
18 FILE_MGR    := %40
19 MEM_MGR     := %00
20
21 EXTERNAL
22 SIGNAL
23 WAIT
24 CREATE_SEG
25 DELETE_SEG
26 SM_SWAP_IN
27 SM_SWAP_OUT
28 TERMINATE
29 MAKE_KNOWN
30
31 ! PAGE .

```


!

```
32 $SECTION FM DATA
33 INTERNAL
34
35 ! * * * * * MESSAGES * * * * * !
36 FM_MSG_1 ARRAY [* BYTE] := '%12FM: IO = SIGNALLER'

37 FM_MSG_2 ARRAY [* BYTE] := '%17FM: CALL KERNEL(CREATE)'

38 FM_MSG_3 ARRAY [* BYTE] := '%16FM: RETURN FROM KERNEL'

39 FM_MSG_4 ARRAY [* BYTE] := '%17FM: CALL KERNEL(DELETF)'
```

40 !PAGE

0000	12	46	4D
0003	3A	20	49
0006	4F	20	3D
0009	20	53	49
000C	47	4E	41
000F	4C	4C	45
0012	52		
0013	17	46	4D
0016	3A	20	43
0019	41	4C	4C
001C	20	4B	45
001F	52	4E	45
0022	4C	28	43
0025	52	45	41
0028	54	45	29
002B	16	46	4D
002E	3A	20	52
0031	45	54	55
0034	52	4E	20
0037	46	52	4F
003A	4D	20	4F
003D	45	52	4E
0040	45	4C	
0042	17	46	4D
0045	3A	20	43
0048	41	4C	4C
004B	20	4B	45
004E	52	4E	45
0051	4C	28	44
0054	45	4C	45
0057	54	45	29


```

!
005A 18 46 4D 41 FM_MSG_5 ARRAY [* BYTE] := '%18FM: CALL KERNEL(SWAP_IN)'
005D 3A 20 43
0060 41 40 40
0063 20 4E 45
0066 52 4E 45
0069 40 28 53
006C 57 41 50
006F 5F 49 4E
0072 29
0073 19 46 4D 42 FM_MSG_6 ARRAY [* BYTE] := '%19FM: CALL KERNEL(SWAP_OUT)'
0076 3A 20 43
0079 41 40 40
007C 20 4E 45
007F 52 4E 45
0082 40 28 53
0085 57 41 50
0088 5F 4F 55
008B 54 29
008D 1A 46 4D 43 FM_MSG_7 ARRAY [* BYTE] := '%1AFM: CALL KERNEL(TERMINATE)'
0090 3A 20 43
0093 41 40 40
0096 20 4E 45
0099 52 4E 45
009C 40 28 54
009F 45 52 4D
00A2 49 4E 41
00A5 54 45 29
00A8 1B 46 4D 43 FM_MSG_8 ARRAY [* PYTE] := '%1BFM: CALL KERNEL(MAKE_KNOWN)'
00AB 3A 20 43
00AF 41 40 40
00B1 20 4E 45
00B4 52 4E 45
00B7 40 28 4D
00BA 41 4E 45
00BD 5F 4E 4E
00C0 4F 57
00C3 29

```


!					
0050	5F00	0178'		CALL PRINT IO IS SIGNALLER	
0054	5F00	00AA'		CALL DELETE_CALL	
0058	5F00	0138'		CALL FM_PRINT_RET_FROM_KER	
				! SIGNAL IO PROC FINISHED!	
005C	5F00	0064'		CALL FM_SIGNAL_CALL	
0060	E8CF			OD !**REPEAT FOREVER**!	
0062	9E0E			RET	
0064				END FM_MAIN	
0064				FM_SIGNAL_CALL	PROCEDURE
0064	6101	00C4'		ENTRY	
0068	760E	00C6'		LD	R1, SENDER
006C	5F00	0000*		LDA	R8, FM_MSG_ARRAY[0]
0070	9E0E			CALL	SIGNAL
0072				RET	
				END FM_SIGNAL_CALL	
0072				WAIT_CALL	PROCEDURE
0072	760E	00C6'		ENTRY	
0076	5F00	0000*		LDA	R8, FM_MSG_ARRAY[0]
007A	6F01	00C4'		CALL	WAIT
007F	9E0E			LD	SENDER, R1 ! SAVE SIGNALING PROCESS # !
0080				RET	
				END WAIT_CALL	

111 !PAGE

!	0080	112	CREATE_CALL	PROCEDURE
		113		
		114	ENTRY	
	0080 2103 0019	115	LD	R3,#25
	0084 5F00 0106	116	CALL	FM_PRINT_BLANKS
	0088 2102 0013	117	LD	R2, #FM_MSG_2
	008C 5F00 015E	118	CALL	FM_PRINT_MSG
	0090 2101 000A	119	LD	R1,#10 !MENTOR SEG #!
	0094 2102 0001	120	LD	R2,#1 !ENTRY #!
	0096 2103 0030	121	LD	R3,#48 !SIZE!
	009C 2104 0003	122	LD	R4,#3 !UPPER WORD OF CLASS!
	00A0 2105 0001	123	LD	R5,#1 !LOWER WORD OF CLASS!
	00A4 5F00 0000*	124	CALL	CREATE_SEG
	00A8 9E08	125	RET	
	00AA	126	END CREATE_CALL	

225

00AA	127	DELETE_CALL	PROCEDURE
	128		
	129		
	130		
	131	ENTRY	
00AA 2103 0019	132	LD	R3,#25
00AE 5F00 0106	133	CALL	FM_PRINT_BLANKS
00B2 2102 0042	134	LD	R2, #FM_MSG_4
00B6 5F00 015E	135	CALL	FM_PRINT_MSG
00BA 2101 000A	136	LD	R1,#10 !MENTOR SEG #!
00BE 2102 0001	137	LD	R2,#1 !ENTRY #!
00C2 5F00 0000*	138	CALL	DELETE_SEG
00C6 9E08	139	RET	
00CE	140	END DELETE_CALL	

!PAGE

!	0104		172	SWAP_IN_CALL	PROCEDURE
			173		
			174	ENTRY	
0104	2103	0019	175	LD R3,#25	
0108	5F07	01C6	176	CALL FM_PRINT_BLANKS	
010C	2102	005A	177	LD R2,#FM_MSG_5	
0110	5F00	015E	178	CALL FM_PRINT_MSG	
0114	2101	000B	179	LD R1,#11 ! SEG #!	
0118	5F00	0000*	180	CALL SM_SWAP_IN	
011C	9E08		181	RET	
011F			182	END SWAP_IN_CALL	
			183		
			184		
			185		
			186		
011E			187	SWAP_OUT_CALL	PROCEDURE
			188		
			189	ENTRY	
011E	2103	0019	190	LD R3,#25	
0122	5F00	01C6	191	CALL FM_PRINT_BLANKS	
0126	2102	0073	192	LD R2,#FM_MSG_6	
012A	5F00	015E	193	CALL FM_PRINT_MSG	
012E	2101	000B	194	LD R1,#11 ! SEG #!	
0132	5F00	0000*	195	CALL SM_SWAP_OUT	
0136	9E08		196	RET	
0138			197	END SWAP_OUT_CALL	

!PAGE


```

198
199
200 FM_PRINT_RET_FROM_KER PROCEDURE
201
202 ENTRY
203 CALL FM_DELAY
204 LD R3,#25
205 CALL FM_PRINT_BLANKS
206 LD R2,#FM_MSG_3
207 CALL WRITELN
208 CALL CRLF
209 LD R3,#75
210 CALL FM_PRINT_LINE
211 CALL CRLF
212 RET
213 END FM_PRINT_RET_FROM_KER
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230

```

```

0138
0139
0140
0141
0142
0143 5F00 0192
0144 2103 0019
0145 5F00 0106
0146 2102 002B
0147 5F00 0FC0
0148 5F00 0FD4
0149 2103 004B
0150 5F00 01AE
0151 5F00 0FD4
0152 9E08
0153
0154
0155
0156
0157
0158
0159
0160
0161
0162 5F00 0FC0
0163 5F00 0192
0164 5F00 0FD4
0165 2103 004E
0166 5F00 01AE
0167 5F00 0FD4
0168 9E0E
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178

```



```

231 PRINT IO IS SIGNALLER
232
233 ENTRY
234 LD R3,#25
235 CALL FM_PRINT_BLANKS
236 LD R2, #FM_MSG_1
237 CALL WRITELN
238 CALL CRLF
239 CALL FM_DELAY
240 RET
241 END PRINT IO IS SIGNALLER

```

```

FM_DELAY PROCEDURE
!*****!
! PRODUCES 2 SEC DELAY !
!*****!

```

```

249 ENTRY
250 LD R2, #COUNT
251 LD R1, #TIME
252 DO
253 CP R2, #0
254 IF EQ THEN EXIT FI
255 DEC R2
256 MREQ R1
257 OD
258 RET
259 END FM_DELAY

```

```

!
0178 0019
017C 01C6
0180 0000
0184 0FC0
0188 0FD4
018C 0192
0190 9E08
0192

0192
0192 2102 000A
0196 2101 01F4
019A 0E02 0000
019E 5E0E 01A6
01A2 5E08 01AC
01A6 AE20
01AE 7B1D
01AA E3F7
01AC 9E0E
01AE

```



```

2 IO_PROCESS      MODULE
3
4 ! VERS 1.8      !
5
6 CONSTANT
7 WRITE           ! TYWR MON CALL !
8 WRITELN         ! PUTMSG MON CALL !
9 CRLF            ! MON CALL      !
10 RETURN_TO_MONITOR := %A902    ! HBUG REENTRY  !
11
12 COUNT := 10
13 TIME := 500
14
15 SPACE := %20
16 DASH := %2D
17
18 IO_MGR      := %20
19 FILE_MGR    := %40
20 MEM_MGR     := %00
21
22 EXTERNAL
23 MAKE_KNOWN
24 TERMINATE
25 SM_SWAP_IN
26 SM_SWAP_OUT
27 SIGNAL
28 WAIT
29 !PAGE

```

```

PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE

```


!

```
0000 10 49 4F
0003 3A 20 52
0006 45 41 44
0009 20 43 4F
000C 4D 41 41
000F 4E 44
0011 0F 49 4F
0014 3A 20 53
0017 49 47 4E
001A 41 4C 20
001D 46 4D 20
0020 20
0021 1B 49 4F
0024 3A 20 43
0027 41 4C 4C
002A 20 4B 45
002D 52 4E 45
0030 4C 28 4D
0033 41 4B 45
0036 5F 4B 4E
0039 4F 57 4E
003C 29
003D 18 49 4F
0040 3A 20 43
0043 41 4C 4C
0046 20 4B 45
0049 52 4E 45
004C 4C 28 53
004F 57 41 50
0052 5F 49 4E
0055 29
```

```
30 $SECTION IO_DATA
31 INTERNAL
32 ! ** * MESSAGES * * * !
33 IO_MSG_1 ARRAY [* BYTE] := '%10IO: READ COMMAND'

34 IO_SEND ARRAY [* BYTE] := '%0FIO: SIGNAL FM '

35 IO_MSG_2 ARRAY [* BYTE] := '%1BIO: CALL KERNEL(MAKE_KNOWN)'

36 IO_MSG_3 ARRAY [* BYTE] := '%18IO: CALL KERNEL(SWAP_IN)'

37 !PAGE
```



```

! 0056 19 49 4F
0059 3A 20 43
005C 41 4C 4C
005F 20 4B 45
0062 52 4E 45
0065 4C 28 53
006E 57 41 50
006B 5F 4F 55
006E 54 29
0070 1A 49 4F
0073 3A 20 43
0076 41 4C 4C
0079 20 4F 45
007C 52 4E 45
007F 4C 28 54
0082 45 52 4D
0085 49 4E 41
0088 54 45 29
008B 16 49 4F
008E 3A 20 52
0091 45 54 55
0094 52 4E 20
0097 46 52 4F
009A 4D 20 4B
009D 45 52 4E
00A0 45
00A2

38 IO_MSG_4 ARRAY [* BYTE] := '%19IO: CALL KERNEL(SWAP_OUT)'

39 IO_MSG_5 ARRAY [* BYTE] := '%1AIO: CALL KERNEL(TERMINATE)'

40 IO_MSG_6 ARRAY [* BYTE] := '%16IO: RETURN FROM KERNEL'

41 IO_MSG_ARRAY ARRAY [16 BYTE]
42 !PAGE

```



```

43 INTERNAL
44
45 $SECTION IO_PROC
46
47
48 IO_MAIN PROCEDURE
49 ENTRY
50 DO !** DO FOREVER **!
51 ! ** PRINT: "READ COMMAND" ** !
52 LD R2, #IO_MSG_1
53 CALL IO_PRINT_MSG
54 ! ** SIGNAL FM TO CREATE ** !
55 ! ** AND WAIT ** !
56 CALL IO_SIGNAL_CALL
57 LDA R8, IO_MSG_ARRAY[0]
58 CALL WAIT
59 LD R2, #IO_MSG_1
60 CALL IO_PRINT_MSG
61 ! ** SIGNAL FM TO MAKE_KNOWN & SWAP_IN ** !
62 ! ** AND WAIT ** !
63 CALL IO_SIGNAL_CALL
64 LDA R8, IO_MSG_ARRAY[0]
65
66 CALL WAIT
67 LD R2, #IO_MSG_1
68 CALL IO_PRINT_MSG
69 ! IO "READ" -- MAKE_KNOWN AND SWAP_IN !
70 CALL IO_MK_CALL
71 CALL IO_SWAP_IN_CALL
72 LD R2, #IO_MSG_1
73 CALL IO_PRINT_MSG
74 ! SIGNAL FM TO SWAP_OUT AND TERMINATE !
75 CALL IO_SIGNAL_CALL
76 LDA R8, IO_MSG_ARRAY[0]
77 ! PAGE

```


! 0048	5F00	0000*	78	CALL	WAIT
004C	2102	0000	79	LD	R2,#IO_MSG_1
0050	5F00	0082	80	CALL	IO_PRINT_MSG
0054	5F00	00D6	81	! IO SWAP_OUT AND TERMINATE !	
0058	5F00	00E8	82	CALL	IO_SWAP_OUT_CALL
005C	2102	0000	83	CALL	IO_TERMINATE_CALL
0060	5F00	0082	84	LD	R2,#IO_MSG_1
0064	5F00	0074	85	CALL	IO_PRINT_MSG
0068	7608	00A2	86	! SIGNAL FM TO DELETE !	
006C	5F00	0000*	87	CALL	IO_SIGNAL_CALL
0070	EC07		88	LDA	RS,IO_MSG_ARRAY[0]
0072	9E08		89	CALL	WAIT
0074			90	OD !**REPEAT FOREVER**!	
			91	RET	
			92	END IO_MAIN	
			93		
			94		
0074			95	IO_SIGNAL_CALL	PROCEDURE
			96		
			97	ENTRY	
0074	2101	0040	98	LD	R1,#FILE_MGR !LD SIGNED PROC #!
0078	7608	0011	99	LDA	RS,IO_SEND[0]
007C	5F00	0000*	100	CALL	SIGNAL
0080	9E08		101	RET	
0082			102	END IO_SIGNAL_CALL	
			103	!PAGE	

LIST OF REFERENCES

1. Coleman, A. R., Security Kernel Design for a Microprocessor-Based, Multilevel, Archival Storage System, MS Thesis, Naval Postgraduate School, December 1979.
2. Conway, T., and others, Introduction to Microprocessors Programming Using PLZ, Winthrop Publishers, 1979.
3. Denning, D.E., "A Lattice Model of Secure Information Flow," Communications of the ACM, v. 19 p. 236-242, May 1976.
4. Gary, A.V. and Moore, E.E., The Design and Implementation of the Memory Manager for a Secure Archival Storage System, MS Thesis, Naval Postgraduate School, June 1980.
5. Madnick, S. E., and Donovan, J.J., Operating Systems, McGraw Hill, 1974.
6. O'Connell, J. S., and Richardson, L. D., Distributed Secure Design for a Multi-microprocessor Operating System, MS Thesis, Naval Postgraduate School, June 1979.
7. Parks, E. J., The Design of a Secure File Storage System, MS Thesis, Naval Postgraduate School, December, 1979.
8. Reed, P. D., and Kanoidia, R. K., "Synchronization With Eventcounts and Sequencers," Communications of the ACM, v. 22 no. 2 p. 115-124, February 1979.
9. Reitz, S. L., An Implementation of Multiprogramming and Process Management for a Security Kernel Operating System, MS Thesis, Naval Postgraduate School, June 1980.
10. Schell, Lt.Col. R. R., "Computer Security: the Achilles Heel of the Electronic Air Force?," Air University Review, v. 30 no. 2 p. 16-33, January 1979.
11. Schell, Lt.Col. R. R., "Security Kernels: A Methodical Design of System Security," USE Technical Papers (Spring Conference, 1979). pp 245-250, March 1979.

12. Snook, T., and others, Report on the Programming Language PLZ/SYS, Springer-Verlag, 1978.
13. Zilog, Inc., Z8000 PLZ/ASM Assembly Language Programming Manual, 03-3055-01, Revision A, April 1979.
14. Zilog, Inc., Z8001 CPU Z8002 CPU, Preliminary Product Specification, March 1979.
15. Zilog, Inc., Z8010 MMU Memory Management Unit, Preliminary Product Specification, October 1979.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4. Lyle A. Cox, Jr., Code 52C1 Department of Computer Science Naval Postgraduate School Monterey, California 93940	4
5. LTCOL Roger R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	5
6. Joel Trimble, Code 221 Office of Naval Research 800 North Quincy Arlington, Virginia 22217	1
7. LCDR John T. Wells P.O. Box 366 Waynesboro, Mississippi 39367	2
8. Office of Research Administration Code 012A Naval Postgraduate School Monterey, California 93940	1
9. Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
10. I. Larry Avrunin, Code 18 DTNSRDC Bethesda, Maryland 20084	1

- | | | |
|-----|--|---|
| 11. | R. P. Crabb, Code 9134
Naval Oceans Systems Center
San Diego, California 92152 | 1 |
| 12. | Kathryn Heninger, Code 7503
Naval Research Lab
Washington, D.C. 20375 | 1 |
| 13. | Dr. J. McGraw
U.C. - L.L.L. (1-794)
P.O. Box 808
Livermore, California 94550 | 1 |
| 14 | Mark Underwood
NPRDC
San Diego, California 92152 | 1 |
| 15. | Walter P. Warner, Code K70
NSWC
Dahlgren, Virginia 22448 | 1 |
| 16. | M. George Michael
U.C. - L.L.L. (L-76)
P.O. Box 808
Livermore, California 94550 | 1 |

Thesis

W441

c.1

Wells

190360

Implementation of segment management for a secure archival storage system.

Thesis

W441

c.1

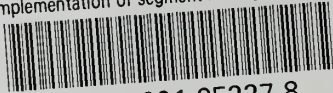
Wells

190360

Implementation of segment management for a secure archival storage system.

thesW441

Implementation of segment management for



3 2768 001 95227 8
DUDLEY KNOX LIBRARY